Boas Práticas: Crie um agendador de tarefas usando técnicas de POO e Padrões de Projeto – Parte 3

Clube Delphi +PHP

DevMedia group

Quick Update
Diretivas de compilação - Sua aplicação compatível com diferentes versões do Delphi

Avançado
O que todo desenvolvedor deve saber sobre Ponteiros

PHP
Persistência em XML com MVC

Ano 9 :: Edição 118 :: R$ 11,90

ISSN 1517990-7

9 771517 990009 00118

DELPHI PRISM

Desenvolva para Web reaproveitando seus conhecimentos em Win32 / Desktop

Brinde

Comprando esta edição ganhe 4 vídeo-aulas:

1 – Criando um componente para máscara e validação de CEP, CPF e CNPJ
2 – Delphi Prism - Web: Comparativo ente VCL e ASP.NET
3 – Delphi Prism - Linguagem LINQ
4 – Delphi Prism - Métodos de extensão

Cursos Online

A **Revista WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade **.Conheça abaixo os cursos já disponíveis.**

Curso de .net em destaque

Aplicação completa de orçamento Doméstico no Visual Studio 2005

Confira neste curso online como criar uma aplicação completa no Visual Studio 2005, usando ASP.NET, Web Services, Mobile e muito mais! Veja como criar uma aplicação de orçamento doméstico, usando diagrama de classes de uma forma muita produtiva, criar classes de negócios de acesso a dados.

Confira o plano de aula completo:
www.devmedia.com.br/domesticovs

Curso de Java em destaque

Introdução ao desenvolvimento para celulares com J2ME

Confira neste curso os principais recursos do J2ME. Aprenda também o passo a passo para criar sua primeira aplicação J2ME. Neste curso você irá aprender diversas funcionalidades desta tecnologia para desenvolvimento de dispositivos móveis.

Confira o plano de aula completo:
www.devmedia.com.br/celularesj2me

Assine a **WebMobile** e comece já seu treinamento!
www.devmedia.com.br/assine

A sua melhor opção de aprendizagem!

Em breve mais novidades para você! www.devmedia.com.br/curso

DevMedia group

Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038

Sumário

Quick Update

7 – Diretivas de compilação
Compile um mesmo código-fonte em várias versões do Delphi
[*Daniel Sobrinho Laporte*]

Delphi — Expert

10 – Ponteiros
Porque precisamos e o que podemos fazer com eles
[*Vitor Luiz Rubio*]

Boas Práticas — Boas Práticas - Expert

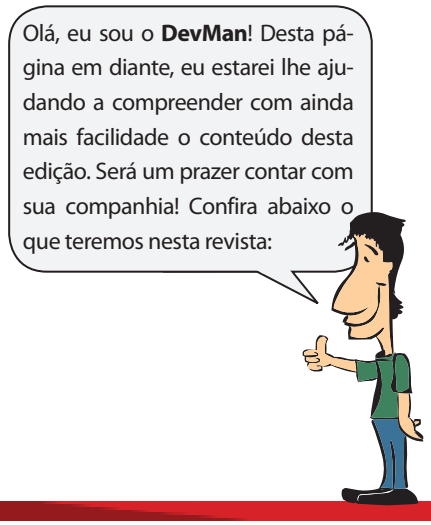
22 – Crie um agendador de tarefas 00 – Parte 3
Framework de persistência
[*Rafael Stavarengo*]

Easy Delphi — Easy Delphi - Mini-Curso – Web - Tutorial

36 – Delphi Prism e ASP.NET
Reaproveite seus conhecimentos Win32 / Desktop para criar aplicações Web – Parte 1
[*Guinther Pauli*]

PHP — PHP - Boas Práticas

42 – XML
Persistência em XML com MVC
[*Lucas Simões Maistro e Raphael Zanon Rodrigues*]



Brinde na web desta edição - 4 vídeos! Confira:

4 Vídeos

- **Criando um componente para máscara e validação de CEP, CPF e CNPJ**
  http://www.devmedia.com.br/post-16384-Curso-Online-Dominando-a-Orientacao-a-Objetos-e-Componentes-em-Delphi-Avancado-Parte-XXIV-Criando-um-Edit-para-Mascarar-e-Validar-CNPJ-CPF-Telefone-Cep-e-Titulo-Parte-1.html

- **Delphi Prism - Web: Comparativo VCL e ASP.NET**
  http://www.devmedia.com.br/post-15638-Curso-Online-Delphi-Prism-Parte-30-Desenvolvimento-Web-Comparativo-VCL-x-ASP-NET-Controles-de-Selecao-ListBox-DropDownList-RadioButtonList-e-CheckBoxList.html

- **Delphi Prism - Linguagem LINQ**
  http://www.devmedia.com.br/post-14243-Curso-Online-Delphi-Prism-Parte-14-Linguagem-LINQ.html

- **Delphi Prism - Métodos de extensão**
  http://www.devmedia.com.br/post-13623-Curso-Online-Delphi-Prism-Parte-9-Metodos-de-Extensao.html

Para acessar os vídeos utilize os seguintes dados
Login: **DVM.PL** *Senha:* **inf8**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.PL** Senha: **inf8**

Clube Delphi +PHP

Ano 9 - 118ª Edição 2010 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Laporte
daniel.laporte@gmail.com

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Capa e Diagramação
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

O romance do Delphi com o .NET foi uma novela, mas que felizmente acabou com um lindo final feliz. O C# Builder foi a primeira ferramenta da Borland que suportou o .NET Framework. O Delphi 7 trouxe um preview que permitia o desenvolvimento de aplicações para .NET usando apenas um compilador, sem IDE. Naquela época (2003), publicamos nossos primeiros artigos sobre a tecnologia (veja meus artigos na edição 37). O C# Builder inaugurou um novo IDE da Borland, chamado BDS, que mais tarde teve suporte ao Delphi na sua versão 2.0 (o Delphi for .NET). O Delphi 8 (veja meu artigo na edição 48) foi o primeiro passo para que os desenvolvedores Delphi pudessem criar aplicações .NET usando tecnologias robustas, como ASP.NET, ADO.NET e Web Services. O Delphi 2005 uniu o Delphi Win32, Delphi for .NET e C# Builder em um único IDE (veja edição 57). Era o BDS 3.0. Depois disso, tivemos o suporte ao .NET 2.0 no RAD Studio 2007 e em 2009 tivemos uma grande mudança de planos, que permanece em 2010. O novo IDE foi construído para incorporar uma série de linguagens em um único ambiente. Porém, para quem desenvolve para o .NET, sempre existiu um problema, grande. Quando a Microsoft lança uma nova versão do .NET Framework, os desenvolvedores Delphi precisam esperar que um novo IDE seja lançado para suportar os novos recursos. Quando a Microsoft lançou o Visual Studio 2005 com suporte ao .NET Framework 2.0, os desenvolvedores Delphi somente 2 anos depois puderem contar com esse suporte, e muito limitado. Depois veio o .NET 3.0, 3.5 e agora o 4.0. A cada nova versão do framework, inúmeros recursos são lançados, novas tecnologias, novos controles, novos namespaces. Depois do .NET Framework 2.0, tivemos várias tecnologias lançadas disponíveis para os desenvolvedores que usam o Visual Studio, C# ou VB.NET, onde podemos destacar: LINQ, WPF, WCF, ASP.NET AJAX, Silverlight, ASP.NET MVC, ADO.NET Entity Framework, WCF Data Services, SharePoint, Windows Azure (Cloud Computing) etc. O grande "Q" da questão é que os desenvolvedores Delphi sempre estavam atrás no que diz respeito ao desenvolvimento com o .NET (e ao crescimento tecnológico como um todo, pois muita coisa gira em volta do .NET), pois dependiam do lançamento de um novo IDE que suportasse uma nova versão do framework. É fato que muitos desenvolvedores Delphi, ao decidirem adotar o .NET Framework, principalmente para soluções Web com ASP.NET, acabaram por adotar o Visual Studio com o C#. Pensando nisso, há alguns anos atrás, a empresa RemObjects, então parceira da Borland, criou o projeto Chrome, uma nova linguagem, uma espécie de evolução do Object Pascal, importando muitas coisas interessantes que o C# e suas novas versões ofereciam. Essa linguagem não usa o BDS como IDE, mas o próprio Visual Studio. O Chrome então é o melhor de dois mundos: uma linguagem clássica e robusta, um Object Pascal turbinado (que pessoalmente eu prefiro chamar de D#) dentro de um IDE já consolidado, estável e produtivo, o Visual Studio (atualmente na versão 2010). O Chrome (concorrente direto do Delphi for .NET), mais tarde veio a se chamar Oxygene, que foi então comprado pela Embarcadero, empresa que adquiriu a CodeGear, levando o intuitivo e comercial nome de Delphi Prism (que pôs fim à marca "Delphi for .NET", ou seja, o .NET dentro do BDS / RAD Studio).

O objetivo da série que iniciamos nesta edição é ensinar você a usar o IDE Delphi Prism / Visual Studio aproveitando seus conhecimentos em BDS / RAD Studio. Aprender D# lembrando do Object Pascal. Desenvolver para ASP.NET se espelhando nos controles da VCL. Criar aplicações Web reaproveitando seus conhecimentos de anos em programação Desktop Win32. Usar o ADO.NET como se fosse o dbExpress. Trabalhar com o SQL Server como se fosse o Firebird.

Ainda nesta edição, confira um excelente artigo do Vitor Rubio sobre ponteiros, a continuação da excelente série do Rafael Stavarengo sobre OO e na sessão PHP o artigo do Lucas Simões e Raphael Zanon sobre XML.

Boa leitura!

Guinther Pauli

guinther@devmedia.com.br

Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix

Microsoft Certified: MCP,MCAD,MCSD.NET,MCTS,MCPD (ASP.NET,C#)

guintherpauli.blogspot.com

twitter.com/guintherpauli

ClubeDelphi PLUS

DevMedia group

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Perguntas e Respostas

Criando colunas virtuais em SQL

Pergunta respondidas por:
Daniel Sobrinho

Gostaria de saber se é possível criar colunas virtuais utilizando apenas instruções SQL e utilizá-las em um dbLookupComboBox como se fossem campos do próprio banco de dados.

Dúvida enviada pelo leitor Giuliano Forgiarini - Belo Horizonte – MG

Para a solução do problema criaremos um banco de dados muito simples. Em nosso caso, demonstrarei através de uma tabela composta por dois campos e quatro registros. Pois bem, utilize uma ferramenta de gerenciamento de bancos de dados como IBExpert ou outra de sua preferência para a criação de um banco dados, em seguida execute as instruções SQL da **Listagem 1** para criar a estrutura proposta.

Listagem 1. Instruções SQL

```
CREATE TABLE PESSOA (
IDPESSOA INTEGER NOT NULL PRIMARY KEY,
PESSOA VARCHAR(100) NOT NULL);

INSERT INTO PESSOA (IDPESSOA, PESSOA) VALUES
 (1, 'DANIEL SOBRINHO LAPORTE');
INSERT INTO PESSOA (IDPESSOA, PESSOA) VALUES
 (2, 'GUINTHER PAULI');
INSERT INTO PESSOA (IDPESSOA, PESSOA) VALUES
 (3, 'PAULO QUICOLI');
INSERT INTO PESSOA (IDPESSOA, PESSOA) VALUES
 (4, 'RODRIGO CARREIRO MOURÃO');
```

Note que o que fizemos foi simplesmente criar uma tabela denominada PESSOA e uma estrutura composta por dois campos (IDPESSOA e PESSOA), inserindo seguidamente quatro registros na mesma.

Abra seu Delphi 2010 e crie uma VCL Forms Application. Insira no formulário os componentes DBLookupComboBox, Button, SQLConnection, SQLDataSet, DataSetProvider, ClientDataSet e DataSource. Renomeie-os respectivamente para dbLookup, btnExecutar, Connection, sql, dsp, cds e dsp. Configure no SQLConnection a conexão com o banco de dados criado anteriormente e faça as ligações informando o SQLConnection à propriedade de mesmo nome do componente SQLDataSet, aponte à propriedade DataSet do DataSetProvider o componente SQLDataSet, à propriedade ProviderName do ClienteDataSet informe o dsp e por último, informe o cds no DataSource. Em seguida, selecione o componente dbLookup e configure sua propriedade ListSource apontando para dsr e informe IDPESSOA em KeyField e PESSOA em ListField. O layout de nossa aplicação você confere na **Figura 1**.

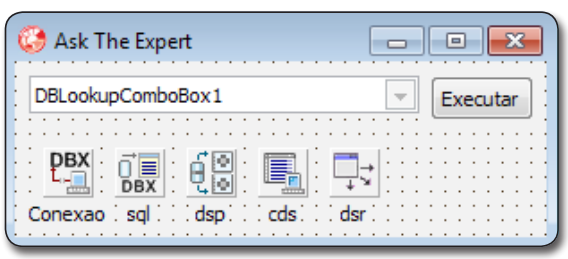


Figura 1. Layout da aplicação

Adicione à propriedade CommandText do SQLDataSet a seguinte instrução:

```
SELECT IDPESSOA, PESSOA FROM PESSOA
```

Em seguida implemente a codificação a seguir no evento onClick do btnExecutar:

```
cds.Close;
cds.Open;
DBLookup.KeyValue := cds.Fields[0].Value;
```

Execute a nossa aplicação e clique no botão Executar. Você perceberá que ainda não possuímos uma coluna virtual, tudo que temos até o momento são as próprias instruções armazenadas no banco de dados. Faremos então uma pequena modificação na SQL deixando-a como a **Listagem 2** sugere.

Listagem 2. SQL modificada

```
SELECT
  IDPESSOA,
  PESSOA
FROM
  PESSOA
UNION
SELECT
  CAST(0 AS INTEGER) AS IDPESSOA,
  CAST('SELECIONE UMA PESSOA' AS VARCHAR(100)) AS PESSOA
FROM
  RDB$DATABASE
```

Como podemos observar na **Listagem 2**, adicionamos uma cláusula UNION à nossa instrução. Esta mesma instrução como o próprio nome sugere, permite-nos uma união com outra tabela qualquer, neste caso adotamos a tabela de sistema RDB$DATABASE que retorna um único registro sempre. Você sempre pode utilizar a cláusula UNION desde que esta união contenha uma estrutura idêntica em quantidade de colunas, e se possível tamanho de campos, para evitar possíveis erros de tentativas de conversão, por exemplo. Desta forma realizamos um CAST (para prevenir este tipo de erro), criando duas colunas virtuais que se referem ao conteúdo que será exibido ao executarmos a aplicação, ou seja, teremos o valor 0 como identificação do código da pessoa e o texto "SELECIONE UMA PESSOA" referente ao campo pessoa. Novamente execute a aplicação e clique no botão Executar, o resultado você confere na **Figura 2**.

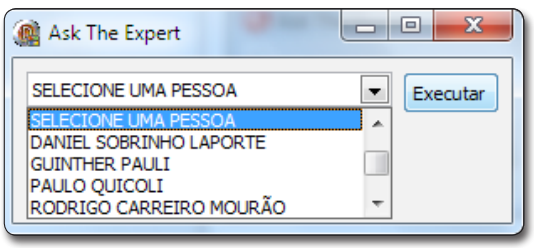


Figura 2. Resultado final da aplicação

Envie suas dúvidas para nossa equipe de especialistas:
guinther@devmedia.com.br
pauloquicoli@devmedia.com.br
daniel.sobrinho@devmedia.com.br
rodrigo.carreiro@devmedia.com.br

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Quick Update:

Diretivas de compilação

Compile um mesmo código-fonte em várias versões do Delphi

Daniel Sobrinho Laporte
daniel.sobrinho@devmedia.com.br
daniel.laporte@gmail.com
www.twitter.com/daniel_laporte
Colunista e editor técnico da Revista ClubeDelphi. Graduado em Sistemas de informação e pós-graduado em Administração de Bancos de Dados pela Universidade de Franca. É analista de sistemas e desenvolvedor Delphi desde 2005 desenvolvendo aplicações cliente/servidor e multicamadas.

Muitos desenvolvedores ainda sofrem ou tem receio de migrar seus projetos para versões mais atuais do Delphi, porque algumas vezes são necessários ajustes no código. É comum escutarmos algo do tipo: "A minha aplicação foi desenvolvida em Delphi 7 e não roda no Delphi 2010". Pois bem, iremos mostrar uma maneira simples de compatibilizar suas aplicações entre diferentes versões utilizando diretivas de compilação. Esta técnica é bem útil e muito usada no desenvolvimento ou adaptação de componentes e evita muitos Warnings do compilador. Neste caso iremos exibir dois exemplos, observe primeiramente a **Listagem 1**.

O código da **Listagem 1** exibe um exemplo simples de uma função que recebe uma instrução qualquer como parâmetro (como um INSERT, DELETE ou UPDATE) para execução direta no banco de dados. Note que temos a declaração de uma variável de tipo TTransactionDesc na linha 04. Esta variável refere-se à própria transação em si. Perceba que setamos um ID e informamos o tipo de ação que a transação tomará (linhas 10 e 11). Em seguida, iniciamos o método StartTransaction do SQLConnection passando como parâmetro a nossa variável transacional já devidamente configurada, para que imediatamente sejam executadas pelo SQLDataSet. Caso a execução seja bem sucedida, será realizado um Commit conforme a linha 15. Caso contrário será verificado o estado da transação no SQLConnection e, em seguida, nossa aplicação executará um Rollback (linha 21).

Compilando este projeto no Delphi 2010 não obteremos nenhum erro do compilador, porém atente à **Figura 1.**

Listagem 1. Rotina de execução de SQL

```
01 function TForm1.SQLExecute(SQL : String): Boolean;
02 var
03  TransactionDesc  :TTransactionDesc;
04  SQLDataSet      :TSQLDataSet;
05 begin
06  SQLDataSet := TSQLDataSet.Create(Nil);
07  SQLDataSet.SQLConnection := SQLConnection1;
08  try
09   try
10    TransactionDesc.TransactionID  := Random(999);
11    TransactionDesc.IsolationLevel := xilREADCOMMITTED;
12    SQLConnection1.StartTransaction(TransactionDesc);
13    SQLDataSet.CommandText := SQL;
14    SQLDataSet.ExecSQL;
15    SQLConnection1.Commit(TransactionDesc);
16    Result := True;
17   except
18    on e: exception do
19    begin
20     if SQLConnection1.InTransaction then
21      SQLConnection1.Rollback(TransactionDesc);
22     Result := False;
23    end;
24   end;
25  finally
26   SQLDataSet.Close;
27   SQLDataSet.Free;
28  end;
29 end;
```

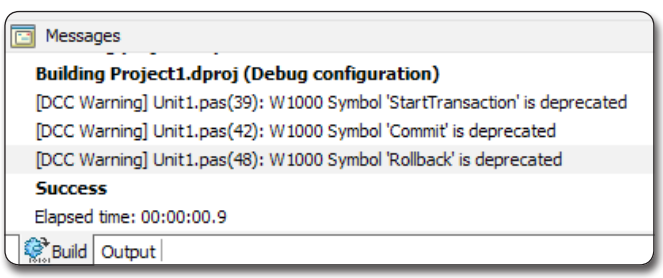


Figura 1. Mensagens de Warning do compilador

A **Figura 1** exibe 3 mensagens de Warning, referindo-se aos métodos StartTransaction, Commit e Rollback. O compilador nos diz aqui que estes métodos estão "ultrapassados" (deprecated). Veremos como resolver estes Warnings compatibilizando com as novas implementações do DBX, permitindo ainda a compilação em suas aplicações anteriores ao Delphi 2010. O primeiro passo para isso é adicionar a unit DBXCommon ao Uses do nosso formulário, porém, adicionando esta unit e tentando abrir o projeto em seu Delphi 7, o compilador não reconhecerá esta unit, gerando assim um erro. Neste caso adaptaremos o uses como a seguir:

```
uses
 Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
 Dialogs, WideStrings, DB, SqlExpr, {$ifdef ver210} DBXCommon {$endif};
```

Perceba que incluímos uma diretiva de compilação {$ifdef ver210}. Com esta diretiva estamos informando ao compilador que caso a versão adotada seja o Delphi 2010, a cláusula uses deve considerar a unit DBXCommon, caso contrário a mesma deve ser ignorada. Faremos as devidas adaptações ao código presente na **Listagem 1.** Observe o código devidamente modificado em nossa **Listagem 2.**

Nota

Do mesmo modo que podemos utilizar a diretiva {$ifdef verXXX} para afirmar que determinado código é pertencente àquela versão específica, podemos utilizar uma diretiva negando esta afirmação, mas sem a necessidade de utilizar o {$else}. Basta definirmos a diretiva como {$ifndef verXXX} que estaremos realizando o processo oposto.

Observando a **Listagem 2** podemos notar que o código original não sofreu alterações, mas adicionamos algumas linhas a mais. Atente que nas linhas 03 o nome original da variável TransactionDesc foi preservado (assim como o SQLDataSet – linha 09), e com a diretiva condicional podemos jogar a responsabilidade da escolha do tipo correto para o compilador. Na linha 16 substituímos o StartTransaction pelo BeginTransaction que atribui uma instância à nossa variável transacional. A outra parte que foi adicionada refere-se à linha 20, que realiza o mesmo processo do método Commit (que era dado como ultrapassado), e por último as modificações no bloco de exceção (entre as linhas 30 e 42). Com esta nova adaptação deixamos de utilizar a condicional InTransaction para verificar o estado transacional e passamos a adotar o método HasTransaction.O método RollBack também foi substituído por RollbackFreeAndNil que possui o mesmo efeito. Com isso realizamos o "update" do nosso código mantendo a compatibilidade com as versões 7 e 2010 neste caso. Compile sua aplicação e note que não teremos mais mensagens de Warning.

Outro caso no qual podemos aplicar esta técnica, é o aviso do compilador relativo a códigos que implementamos e que podem ser atualizados ou melhorados, ou também compatibilizadas com as novas versões. Para ilustrarmos este exemplo, adicione à sua aplicação um novo Edit e em seu evento OnKeyPress digite o código a seguir:

```
if not (Key in ['0'..'9', #8, #13]) then
 Key := #0;
```

A implementação anterior é uma velha conhecida nossa, utilizada por muitos desenvolvedores para permitir apenas a digitação de valores numéricos e as teclas backspace e enter sequencialmente. Compilando nosso código veja a mensagem que o compilador nos exibe de acordo com a **Figura 2.**

Listagem 2. Função SQLExecute modificada

```
01  function TForm1.SQLExecute(SQL : String): Boolean;
02  var
03   TransactionDesc :
04    {$ifdef ver210}
05     TDBXTransaction
06    {$else}
07     TTransactionDesc
08    {$endif};
09   SQLDataSet :TSQLDataSet;
10  begin
11   SQLDataSet := TSQLDataSet.Create(Nil);
12   SQLDataSet.SQLConnection := SQLConnection1;
13   try
14    try
15     {$ifdef ver210}
16      TransactionDesc := SQLConnection1.BeginTransaction;
17     {$else}
18      TransactionDesc.TransactionID  := Random(999);
19      TransactionDesc.IsolationLevel := xilREADCOMMITTED;
20      SQLConnection1.StartTransaction(TransactionDesc);
21     {$endif}
22       SQLDataSet.CommandText := SQL;
23       SQLDataSet.ExecSQL;
24     {$ifdef ver210}
25       SQLConnection1.CommitFreeAndNil(TransactionDesc);
26     {$else}
27      SQLConnection1.Commit(TransactionDesc);
28     {$endif}
29     Result := True;
30    except
31     on e: exception do
32     begin
33      {$ifdef ver210}
34       if SQLConnection1.HasTransaction(TransactionDesc) then
35        SQLConnection1.RollbackFreeAndNil(TransactionDesc);
36      {$else}
37       if SQLConnection1.InTransaction then
38        SQLConnection1.Rollback(TransactionDesc);
39      {$endif}
40      Result := False;
41     end;
42    end;
43   finally
44    SQLDataSet.Close;
45    SQLDataSet.Free;
46   end;
47  end;
```

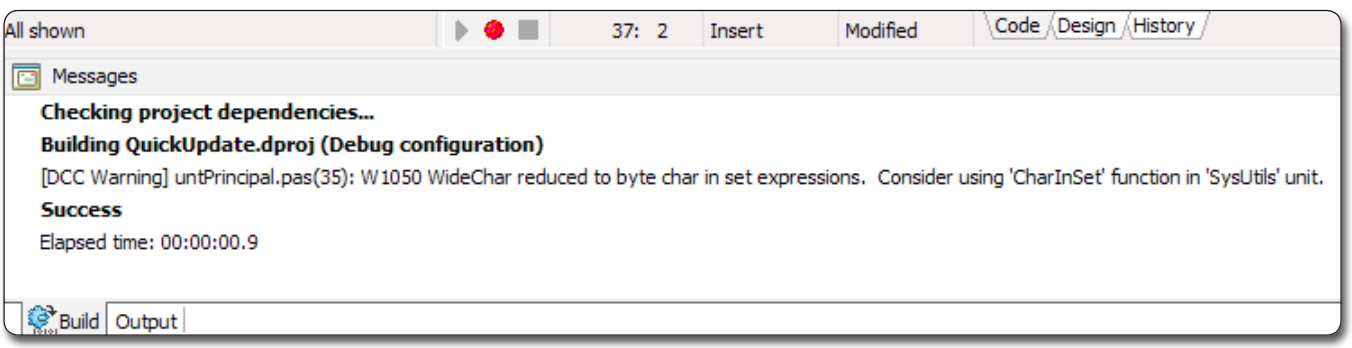


Figura 2. Sugestão do compilador

A mensagem da **Figura 2** nos informa que o que foi codificado pode ser "atualizado" pela função CharInSet, que fará o mesmo efeito e está disponível na unit SysUtils. Seguindo a sugestão do compilador, volte ao evento OnKeyPress e agora modifique-o como a seguir:

```
{$ifdef ver210}
  if not CharInSet(Key, ['0'..'9', #8, #13]) then
{$else}
  if not (Key in ['0'..'9', #8, #13]) then
{$endif}
  Key := #0;
```

Na código anterior utilizamos novamente as diretivas de compilação para atualizar a nossa aplicação. Perceba que ao compilar o projeto nenhum Warning será exibido e sua aplicação poderá ser aberta normalmente no Delphi 7 e 2010. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos intermediários sobre Delphi Win 32 e Delphi for .NET

Ponteiros

Porque precisamos e o que podemos fazer com eles

Do que trata o artigo

Neste artigo abordaremos ponteiros: sua sintaxe atual e mudanças ao longo das versões do Delphi, como e onde utilizá-los e sua evolução. Também abordaremos porque ainda hoje é importante conhecermos ponteiros, quais tipos usados são classificados como ponteiros e como funciona a alocação dinâmica de memória.

Para que serve

Para programar usando a API do Windows, desenvolver componentes, integrar com sistemas legados ou feitos em outras linguagens. Os ponteiros são a base para a alocação dinâmica de memória, e consequentemente os ambientes gerenciados trazem, em seu código fonte, um extenso uso de ponteiros.

Em que situação o tema é útil

Mesmo no Windows 7 a sua API ainda contém vários métodos cujos argumentos ou resultados são ponteiros. E neste nível dificilmente teremos programação orientada a objetos pura, ou um ambiente gerenciado. Estamos falando de um nível intermediário entre a sua aplicação e o sistema operacional. Sempre que tivermos um método em uma DLL escrita em C ou C++, o uso de ponteiros se fará necessário para que possamos passar variáveis por referência.

Resumo do DevMan

Ponteiros são endereços da memória e podem ser armazenados em variáveis. Não são valores, mas apontam para uma área da memória onde há um valor armazenado. Embora esse assunto possa parecer muito acadêmico, é ainda muito utilizado para se aproveitar ao máximo a API do Windows e até se desenvolver coisas novas, como novos componentes que fazem uso da API. Veremos neste artigo o básico de ponteiros, como eles podem ser usados para se passar variáveis por referência e qual sua relação com vetores.

Mesmo nas linguagens de "alto nível" como Delphi, ponteiros são o assunto de nível mais baixo que podemos tratar. Isso porque estaremos lidando diretamente com endereços da memória. Há quem defenda que ponteiros são coisa do passado e que não devem mais ser usados. Mas isso é um engano. Uma rápida olhada no código–fonte da VCL e da RTL pode revelar o tamanho uso que as próprias bibliotecas do Delphi fazem.

Vitor Luiz Rubio
vitorrubio@gmail.com
Analista de Sistemas Sr. na Editora Revista dos Tribunais. Trabalha com Delphi desde a versão 3. Formado em Processamento de Dados pela FATEC-SP.

E mesmo em ambientes gerenciados como Python, Java, Ruby e .NET, embora não usemos ponteiros diretamente ou explicitamente, a máquina virtual, geralmente escrita em C/C++, faz um extenso uso de ponteiros. No próprio Delphi, todo objeto, cuja memória é alocada dinamicamente, é um ponteiro, como veremos mais adiante.

Para armazenar valores nós declaramos variáveis. Declarar uma variável significa reservar um espaço na memória onde poderemos armazenar um valor qualquer. Agora imagine que cada área da memória contém um endereço numérico. Ponteiros trabalham diretamente com os próprios endereços. Ou seja, você pode acessar um valor através de seu endereço na memória em vez de utilizar uma variável.

No Delphi utilizamos o operador ^ (circunflexo) para declarar uma variável do tipo ponteiro. Usamos também o operador @ precedendo variáveis "normais" para obter o endereço delas. O inverso de @ seria usar ^ depois de uma variável-ponteiro, para que possamos acessar o seu valor. Um ponteiro recém declarado que não aponta para nenhum lugar tem o valor nil.

Antes de iniciarmos a nossa aplicação, atente para a **Figura 1** que exibe um exemplo do funcionamento de um ponteiro.

Vamos desenvolver uma aplicação Delphi e ver na prática como tudo isso funciona (no Delphi crie um projeto Win32). Criaremos um botão para cada exemplo, nomeados como btEx<yy> onde yy será o número do nosso exemplo. Os captions de cada botão serão análogos, como "Ex 01", "Ex 02" e assim por diante. Modifique a propriedade Name do formulário para frmPonteiros e informe o Caption "Trabalhando com Ponteiros", por último salve sua unit como uPonteiros.pas e seu projeto como Ponteiros.dproj.

Declarando ponteiros

No nosso primeiro exemplo vamos declarar um ponteiro para integer. Isso significa que teremos um endereço de memória apontando para um local onde será armazenado um valor inteiro. Para ver como funciona o mecanismo de ponteiros iremos declarar duas variáveis do tipo integer e obter o endereço de cada uma, e posteriormente o valor.

Não podemos esquecer de inicializar o nosso ponteiro com nil ou já com um endereço válido, caso contrário ele virá com um endereço randômico de memória, provavelmente apontando para algum "lixo" ou até mesmo para a área de instruções. Esquecer de inicializar ponteiros pode causar acessos indevidos a áreas protegidas de memória e até a corrupção de dados e instruções. Adicione ao formulário o primeiro botão e digite o código da **Listagem 1**.

Listagem 1. Declaração de ponteiros

```
procedure TfrmPonteiros.btEx01Click(Sender: TObject);
var
  variavel1: integer;
  variavel2: integer;
  ponteiro: ^integer;
begin
  //Atribuindo um valor às variáveis
  variavel1 := 123;
  variavel2 := 246;
  //inicializando o ponteiro com nil
  ponteiro := nil;
  if ponteiro = nil then
    ShowMessage('O ponteiro ainda não aponta para lugar nenhum');
    ShowMessage('O endereço de memória da variável 1 é:' +
    inttostr(integer(@variavel1)) );
    ShowMessage('O endereço de memória da variável 2 é:' +
    inttostr(integer(@variavel2)) );  ponteiro := @variavel1;
    ShowMessage('O endereço atual do ponteiro é:' + inttostr
    (integer(ponteiro)) );
    ShowMessage('O valor que ele aponta é:' +inttostr
    (integer(ponteiro^)) );
    ponteiro := @variavel2;
    ShowMessage('O endereço atual do ponteiro é:' + inttostr
    (integer(ponteiro)) );
    ShowMessage('O valor que ele aponta é:' +  inttostr
    (integer(ponteiro^)) );
  //mostrando os tamanhos
    ShowMessage('O tamanho da variavel 1 é:' +   inttostr
    (sizeof(variavel1)) );
    ShowMessage('O tamanho da variavel 2 é:' + inttostr
    (sizeof(variavel2)) );
    ShowMessage('O tamanho do ponteiro é:' + inttostr
    (sizeof(ponteiro)) );
end;
```

Os ponteiros que estamos usando aqui são "tipados", ou seja, são ponteiros para um tipo definido, no caso integer. As três últimas linhas da listagem mostram o tamanho das variáveis, em bytes. As variáveis 1 e 2 são do tipo integer, por isso possuem 4 bytes. O ponteiro também possui 4 bytes, porém, todo ponteiro é, no fundo, um integer, e na arquitetura de 32 bits um ponteiro também tem 32 bits, ou seja, 4 bytes. Experimente trocar o tipo da variável de integer para byte e verá que enquanto as variáveis 1 e 2 tiverem 1 byte, o ponteiro continuará tendo 4 bytes.

Na **Listagem 2** faremos um código similar, mas desta vez mudaremos o valor da variável e verificaremos qual será o valor para o qual o ponteiro aponta. Novamente insira um botão no formulário para a inclusão deste novo código.

Na **Listagem 2** podemos ver que tanto faz referenciar um local na memória através de uma variável ou de um ponteiro,

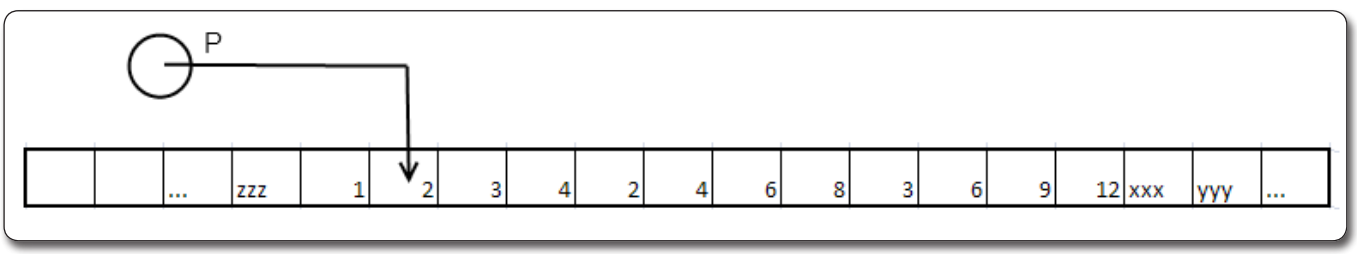


Figura 1. Um ponteiro apontando para um determinado endereço na memória, que contém o valor 2

ambos apontarão para o mesmo local, e caso se mude esse valor, o valor atual será visível em quaisquer ponteiros que apontem para ele. Podemos ter vários ponteiros "apontando" ou referenciando um mesmo lugar na memória.

Listagem 2. Operações com os valores apontados

```
procedure TfrmPonteiros.btEx02Click(Sender: TObject);
var
 variavel: integer;
 ponteiro: ^integer;
begin
 variavel := 50;
 ShowMessage('Valor da variável: ' + IntToStr(variavel));
 //atribuindo ao ponteiro o endereço da variável
 ponteiro := @variavel;
 ShowMessage('Valor apontado pelo ponteiro: ' + IntToStr(ponteiro^));
 variavel := variavel + 10;
 ShowMessage('Valor apontado pelo ponteiro após a soma: ' +
  IntToStr(ponteiro^));
 ponteiro^ := ponteiro^ + 5;
 ShowMessage('Valor atual da variável: ' + IntToStr(variavel));
end;
```

Ponteiros e a passagem de parâmetros

Como sabemos, o Delphi possui a palavra reservada "var" para se passar a um método parâmetros por referência. Isso significa que não será usada dentro do método uma cópia da variável, mas sim a própria variável, podendo ter seu valor alterado.

Crie os métodos da **Listagem 3** e em seguida a chamada a eles no botão 3. Como você pode notar, o código do botão 3 mostra como uma variável pode ter seu valor alterado ao ser passada a um método por referência.

A linguagem C não possui nenhuma palavra reservada para passagem de parâmetros por referência. Uma variável ou estrutura passada a um método seria inteiramente copiada dentro do escopo local do método. Uma prática muito comum no C/C++ para a passagem de parâmetros por referência é passar um ponteiro para a variável. Dentro do método será alterada a área para onde esse ponteiro aponta.

Parece redundância dizer isso, mas no C, você não passa uma variável por referência, e sim uma referência (ponteiro) a uma variável. A diferença está na sintaxe, mas na prática é a mesma coisa.

O terceiro método da **Listagem 3** mostra como a passagem de um ponteiro a um método pode ser equivalente à passagem de uma variável por referência.

Na execução dos métodos mostraremos mensagens exibindo o valor das variáveis em cada etapa. Você pode usar um memo para isso, ou, mais corretamente, uma Watch (janela do debugger, veja **Nota do DevMan**).

Como não podemos declarar dentro dos parâmetros de um método um ponteiro usando explicitamente o operador ^, declaramos o parâmetro como sendo do tipo PInteger. O PInteger nada mais é do que um ponteiro para integer, definido em System.pas como:

```
PInteger = ^Integer;
```

Nota

Você pode inclusive definir um Type, chamado PMyInteger = ^Integer.

Feito isso podemos usar, para alimentar esse parâmetro, qualquer endereço de um integer obtido com o operador @.

Listagem 3. Relação entre ponteiros e passagem de argumentos por referência

```
procedure IncrementaPorValor(i: integer);
begin
 i := i + 1;
 ShowMessage('Valor da variável dentro do método :'+IntToStr(i));
end;

procedure IncrementaPorReferencia(var i: integer);
begin
 i := i + 1;
 ShowMessage('Valor da variável dentro do método :'+IntToStr(i));
end;

procedure IncrementaComPonteiro(i: PInteger);
begin
 i^ := i^ + 1;
 ShowMessage('Valor da variável dentro do método :'+IntToStr(i^));
end;

procedure TfrmPonteiros.btEx03Click(Sender: TObject);
var
 x,y,z: integer;
begin  x := 11;
 y := 22;
 z := 33;
 ShowMessage('Valor da variável x :'+IntToStr(x));
 ShowMessage('Valor da variável y :'+IntToStr(y));
 ShowMessage('Valor da variável z :'+IntToStr(z));
 IncrementaPorValor(x);
 //sabemos que não funcionará
 IncrementaPorReferencia(y); //o método mais conhecido por nós
 IncrementaComPonteiro(@z);  //passando o endereço de z, @z
 ShowMessage('Valor da variável x :'+IntToStr(x));
 ShowMessage('Valor da variável y :'+IntToStr(y));
 ShowMessage('Valor da variável z :'+IntToStr(z));
end;
```

Isso é muito importante caso se usem DLL's feitas em outras linguagens, onde serão passados valores esperando que sejam modificados. A passagem de ponteiros por referência, usando-se a palavra reservada var, pode ser perigosa, pois cria na verdade uma referência duplamente indireta, ou um ponteiro para ponteiro. Isso pode permitir que os endereços de alguns ponteiros sejam trocados, fazendo com que apontem para outros valores, regiões inválidas da memória ou para lugar nenhum. Como era esperado o método IncrementaPorValor incrementará a variável y apenas dentro do método, e não fora dele.

Nota do DevMan

Para usar Watches clique em View > Debug Windows > Watches e a janela de Watches irá aparecer. Essa janela é muito importante para o debug porque com ela você pode visualizar os valores de variáveis durante o processo de debug. Para isso, basta copiar e colar uma variável para dentro da janela ou, com um duplo clique na janela, escrever o nome da variável.

Ao se trabalhar com estruturas de dados muito grandes, ou grandes vetores e matrizes, a passagem de parâmetros por referência é muito mais rápida, pois os dados não são copiados para uma variável interna do método. Mas assegure-se de fazer isso apenas em procedimentos que não alterem o valor ou valores da estrutura.

No estudo ou desenvolvimento de uma linguagem é muito importante saber sobre o efeito das palavras reservadas usadas nos argumentos de métodos, e como esses argumentos são empilhados, onde a memória é alocada etc. Isso é o funcionamento interno de qualquer linguagem que venhamos a trabalhar.

Nota

Por isso nossos métodos não devem aceitar objetos por referência, a menos que eles sejam criados ou destruídos por esses métodos, como no caso do método FreeAndNil, que além de destruir um objeto atribui nil a sua variável.

Como veremos mais adiante, esse é o mistério envolvendo os objetos: suas variáveis são ponteiros e apontam para um objeto criado na memória. Porém nada, nenhuma variável, é o objeto propriamente dito.

Ponteiros e arrays

Arrays no Delphi nada mais são do que ponteiros para um primeiro elemento de uma série de tamanho finita. Para provar isso vamos declarar um array estático global, de 10 posições, do tipo integer. Os códigos das **Listagens 4** e **5** mostrarão como o ponteiro para um array é na verdade o ponteiro para o primeiro elemento deste e mostra como podemos acessar os elementos de um array através de seu ponteiro.

Também veremos um pouco sobre aritmética de ponteiros. Podemos incrementar, decrementar ou somar valores em ponteiros para avançar ou retroceder "n" elementos na memória. As procedures inc e dec podem ser úteis para incrementar ou decrementar um ponteiro. Essas procedures já verificam de que "tipo" o ponteiro é e somam o número de bytes necessários para avançar ou retroceder um elemento.

Isso é importante, porque embora possamos converter um ponteiro para um integer, não basta apenas somar ou subtrair unidades para avançar ou retroceder elementos. Deve-se somar ou subtrair o número de bytes correspondente ao tamanho do tipo de dado que estamos manipulando. Por exemplo, se estivermos trabalhando com byte podemos somar ou subtrair 1. Se estivermos trabalhando com caracteres Unicode devemos somar ou subtrair 2. Trabalhando com integers devemos somar ou subtrair 4, e assim por diante.

Nota

Nunca assuma que o tamanho de um tipo, em bytes, é conhecido. Use sempre a função SizeOf para retornar o tamanho e multiplicar pelo número de elementos que você quer avançar ou retroceder. Por exemplo, em versões antigas do Delphi / Pascal, um integer tinha 16 bits, o mesmo integer hoje tem 32 bits.

A **Listagem 4** mostra como você pode incrementar um ponteiro com inc ou usando uma soma comum. É importante salientar que você pode "extrapolar" os limites de um array e sair "vasculhando" a memória inteira, como se fosse um vetor enorme, e com isso até encontrar endereços onde são alocados dados "interessantes". Lógico que isso não deve ser feito num programa "de verdade", em produção. Isso pode causar muitos "Access Violations", entre outros problemas.

Listagem 4. Relação entre ponteiros e vetores

```
var
  frmPonteiros: TfrmPonteiros;
  vetor: array[1..10] of integer;

implementation

procedure InicializaVetor();
var
  I: Integer;
begin
  //preencheremos os 10 valores do Vetor com a tabuada do 5, de 5 a 50
  for I := 1 to 10 do
    vetor[I] := i*5;
end;

procedure TfrmPonteiros.btEx04Click(Sender: TObject);
var
  p: PInteger;
begin
  InicializaVetor();
  p := @vetor;
  showmessage(inttostr(p^)); //elemento 1 - valor 5
  inc(p);
  showmessage(inttostr(p^)); //elemento 2 - valor 10
  inc(p, 2);
  showmessage(inttostr(p^)); //elemento 4 - valor 20
  p := pointer(integer(p)+SizeOf(integer));
  showmessage(inttostr(p^)); //elemento 5 - valor 25
  p := pointer(integer(p)+SizeOf(integer)*2);
  showmessage(inttostr(p^)); //elemento 7 - valor 35
  p := pointer(integer(p)+SizeOf(integer)*5);
  showmessage(inttostr(p^)); //elemento 12 - valor ???
end;
```

O que fizemos na **Listagem 4** foi criar um vetor de 10 posições, de 1 a 10 e uma procedure para inicializá-lo. Usamos o operador @ para obter o endereço do vetor e com isso descobrimos que o endereço de um vetor é na verdade o endereço de seu primeiro elemento, como comentei. O Delphi se encarrega automaticamente de alocar a memória necessária para armazenar todos os elementos. E o próprio Delphi sabe onde o vetor começa e termina, só assim ele consegue disparar a exceção "Index Out of Bounds" caso você utilize um índice inválido.

A cada linha somamos valores no ponteiro para avançar uma posição no array. Podemos somar com inc ou usando a linha de código:

```
p := pointer(integer(p)+SizeOf(integer)*<<Número a deslocar>>);
```

Isso faz com que convertamos p para integer e somemos o valor do tamanho de um integer multiplicado pelo número de elementos que desejamos navegar. O inteiro resultante nós convertemos de volta para ponteiro.

É muito importante não somar ao ponteiro números "quebrados". Por exemplo, se estamos trabalhando com integers não podemos somar 1 ao ponteiro. Isso fará com que o ponteiro se desloque apenas um byte e tente obter um valor inteiro de uma região na memória que pega o byte mais significativo de um inteiro com os três bytes menos significativos de outro. Isso resultará num número enorme, e totalmente incorreto.

Temos outros tipos de arrays que também devem ser observados. Os arrays multidimensionais, ou matrizes, são multidimensionais apenas do ponto de vista do programador. Do ponto de vista do sistema todos os arrays têm uma única dimensão porque a memória tem uma única dimensão. A memória é um gigantesco array unidimensional. Na **Listagem 5** vamos criar um array de duas dimensões e percorrê-lo através de um ponteiro.

Listagem 5. Relação entre ponteiros e matrizes

```
var
  MatrizBiDimensional: array[1..3, 1..4] of integer;

implementation

procedure InicializaMatrizBiDimensional();
var
  i,j: integer;
begin
  for I := 1 to 3  do
    for j := 1 to 4 do
      MatrizBiDimensional[i,j] := i*j;
end;

procedure TfrmPonteiros.btEx05Click(Sender: TObject);
var
  p: PInteger;
  i: integer;
begin
  InicializaMatrizBiDimensional();
  p := @MatrizBiDimensional;
  for I := 1 to 12 do
  begin
    showmessage(inttostr(p^)); //elemento 1 - valor 5
    inc(p);
  end;
end;
```

Na **Figura 2** mostramos a variável MatrizBiDimensional na Watch List. Repare que um vetor de 3 linhas e 4 colunas de inteiros é transformado em três "vetores" de 4 posições cada um. Podemos ver também o endereço atual de p e o valor, no caso 2, porque essa imagem foi extraída durante a segunda iteração do laço for.

Nota do DevMan

Arrays multidimensionais são o que comumente conhecemos pelo conceito de Matrizes. Ao contrário dos arrays comuns (vetores) que possuem apenas uma direção e acesso simplificado através de um índice, determinados pela linha, os arrays multidimensionais trabalham com linhas e colunas, neste caso necessitando de mais de um índice para acesso de seus valores.

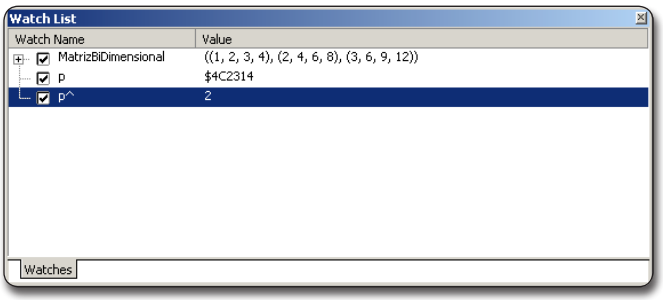


Figura 2. A matriz, o endereço na memória e o valor armazenado no endereço

Como sabemos de antemão que nossa matriz de duas dimensões tem doze elementos, afinal, 3 x 4 = 12, o que fizemos foi obter o ponteiro para o primeiro elemento e varrer linearmente essa região da memória. Usamos o método inc para incrementar corretamente os ponteiros.

Relação entre ponteiros, ShortStrings e PChar

Não podemos falar desse assunto sem antes retroceder um pouco no tempo. Nos primórdios da programação Pascal/Delphi as strings poderiam ter um tamanho máximo de 255 caracteres, e eram formadas por caracteres ASCII, ou seja, caracteres de um byte.

Conforme a evolução do Object Pascal / Delphi Language, surgiram as strings de tamanho "infinito". Na verdade a capacidade desse tipo de string não é realmente infinita, mas o máximo número de caracteres que ela pode ter corresponde ao máximo de um inteiro sem sinal. Um pouco mais de quatro bilhões de caracteres.

Nas últimas versões do Delphi as strings padrão passaram a ser strings Unicode de dois bytes. Ainda existem as strings de caracteres de um byte. São as AnsiStrings, formadas por AnsiChars. PChar (PAnsiChar no nosso caso) é muito usado para se comunicar com a API do Windows porque trata-se de uma string aberta terminada em #0 (como no C/C++).

Na linguagem C uma string na verdade é um ponteiro para char, criando um vetor de tamanho indeterminado. Como fazer para saber onde a string termina e qual o seu tamanho? As strings no C são terminadas por um caractere nulo, #0 para ASCII ou #0#0 para Unicode. Esses ponteiros devem ter a sua memória alocada dinamicamente e o último caractere deve ser um #0. Há uma outra maneira de criar strings desse tipo que é usando arrays de chars.

Nas **Listagens 6** e **7** vamos criar shortstrings, que são aquelas strings que usávamos antigamente no tempo do Pascal, com

tamanho limitado. Usaremos também vetores de Char e ponteiros e faremos chamadas a alguns métodos da API do Windows.

Listagem 6. Relação entre ponteiros e shortstrings

```
procedure TfrmPonteiros.btEx06Click(Sender: TObject);
var
 //shortstrings já são formadas por AnsiChar, por padrão
 UmaString: string[20];
 //como o novo Char agora é WideChar, usamos AnsiChar explicitamente
 UmVetor: array[1..20] of AnsiChar;
 h: HWND;
 PonteiroChar: PAnsiChar; //Equivale a ^AnsiChar
begin
 UmaString := 'ClubeDelphi';
 UmVetor[1] := 'D';
 UmVetor[2] := 'e';
 UmVetor[3] := 'v';
 UmVetor[4] := 'M';
 UmVetor[5] := 'e';
 UmVetor[6] := 'd';
 UmVetor[7] := 'i';
 UmVetor[8] := 'a';
 UmVetor[9] := #0;
 h := FindWindowA(nil,'Trabalhando com Ponteiros'); //teste também
  com 'Calculadora'
 SetWindowTextA(h, UmVetor);
 ShowMessage('Repare que o caption desta janela mudou para: ' +
   AnsiString(UmVetor) );
 SetWindowTextA(h, UmaString);
 ShowMessage('Repare que o caption desta janela mudou para: ' +
   UmaString ); PonteiroChar := @UmVetor; SetWindowTextA
   (h, PonteiroChar);
 ShowMessage('Repare que o caption desta janela mudou para: ' +
   PonteiroChar );
end;
```

O que fizemos na **Listagem 6** foi criar uma ShortString (string de tamanho definido) de 20 caracteres com o conteúdo "ClubeDelphi". Criamos também um vetor de chars com os nove primeiros elementos formando a palavra DevMedia e terminando com um #0. Mostramos como podemos usar livremente tanto a string quanto o vetor ao chamar a função SetWindowTextA da API do Windows. Essa função muda o texto de uma janela encontrada, dado o seu handle. Encontramos o handle de nossa janela com a função FindWindowA e passando como argumento o Caption do nosso próprio form. Podemos também usar o ponteiro para o vetor.

Nota

Handle, na API do Windows, é um número que aponta para uma janela, controle, arquivo, objeto. Imagine um Handle como se fosse um tipo "TObject" do Delphi, ou Pointer, ou seja, pode apontar para qualquer coisa.

Uma questão interessante é que embora o tipo PAnsiChar seja definido como sendo ^AnsiChar, você não pode criar um ponteiro do tipo ^AnsiChar e usá-lo no método SetWindowTextA, porque ela foi definida, em seu cabeçalho, como usando PAnsiChar, e o Delphi considera PAnsiChar um tipo, e controla rigidamente isso.

Tudo o que testamos aqui para AnsiChar e PAnsiChar vale também para Char e PChar, mas como no Delphi o tipo Char agora é Unicode, a API usada deveria ser SetWindowTextW ou apenas seu alias SetWindowText, sem o "A" no final, que corresponde a AnsiChar.

Um aspecto interessante sobre as ShortStrings, ou strings de tamanho fixo, é que antes do primeiro char, em sua posição "zero", é armazenado o tamanho da string. O Delphi gerencia automaticamente a cópia de strings, a limitação do tamanho e a concatenação do #0 no final.

Quando usamos, concatenamos, exibimos ou atribuímos strings o Delphi conta e armazena a quantidade de caracteres no primeiro byte (ou Word) e armazena os caracteres nas posições subsequentes. O compilador faz com que esse primeiro byte fique invisível para nós e opera apenas com os posteriores. Precisamos de ponteiros para ver o primeiro byte.

Obter o tamanho de uma string assim é mais rápido do que contar seus caracteres até encontrar um #0, e é assim que a função Length funciona internamente. Inclusive, o valor que você vai obter nesse byte corresponde ao que a função Length retornaria.

Para provar que uma string desse tipo é um vetor, usamos a função SizeOf, para verificar seu tamanho em bytes, e a função retorna 21, porque a string tem 20 caracteres (bytes) mais o primeiro byte que é o contador de caracteres.

O código da **Listagem 7** mostrará como obter o tamanho de uma string usando seu endereço, apontando para o primeiro elemento e convertendo para byte. Usamos aqui um PByte que nada mais é do que um ^Byte.

Listagem 7. Obtendo o primeiro byte e o valor armazenado nele

```
procedure TfrmPonteiros.btEx07Click(Sender: TObject);
var
 UmaString: string[20];
 comprimento: integer;
begin
 UmaString := 'ClubeDelphi';
 comprimento := PByte(@UmaString)^;
 ShowMessage('Comprimento usando o primeiro byte: ' +
   IntToStr(comprimento));
 ShowMessage('Comprimento usando Length: ' +  IntToStr
   (Length(UmaString)));
 UmaString := '*Clube Delphi*';
 comprimento := PByte(@UmaString)^;
 ShowMessage('Comprimento usando o primeiro byte: ' +
   IntToStr(comprimento));
 ShowMessage('Comprimento usando Length: ' + IntToStr
  (Length(UmaString)));
 UmaString := 'Delphi';
 comprimento := PByte(@UmaString)^;
 ShowMessage('Comprimento usando o primeiro byte: ' +
   IntToStr(comprimento));
 ShowMessage('Comprimento usando Length: ' + IntToStr
  (Length(UmaString)));
 ShowMessage('Tamanho total usando SizeOf: ' +
   IntToStr(SizeOf(UmaString)));
end;
```

Cada vez que alteramos a string obtemos novamente o valor armazenado no primeiro byte, apontado pelo próprio endereço da string. Uma outra maneira de se fazer isso é como mostrado na **Listagem 8**, porém não colocaremos um botão no formulário para fazer dessa forma.

Listagem 8. Outra maneira, usando diretamente o ponteiro

```
procedure TfrmPonteiros.btEx07Click(Sender: TObject);
var
  UmaString: string[20];
  pcomprimento: PByte;
begin
  pcomprimento := @UmaString;
  UmaString := 'ClubeDelphi';
  ShowMessage('Comprimento usando o primeiro byte: ' +
    IntToStr(pcomprimento^));
  ShowMessage('Comprimento usando Length: ' +    IntToStr
   (Length(UmaString)));
  UmaString := '*Clube Delphi*';
  ShowMessage('Comprimento usando o primeiro byte: ' +
    IntToStr(pcomprimento^));
  ShowMessage('Comprimento usando Length: ' +    IntToStr
    (Length(UmaString)));
  UmaString := 'Delphi';
  ShowMessage('Comprimento usando o primeiro byte: ' +
    IntToStr(pcomprimento^));
  ShowMessage('Comprimento usando Length: ' + IntToStr
    (Length(UmaString)));
  ShowMessage('Tamanho total usando SizeOf: ' +
    IntToStr(SizeOf(UmaString)));
end;
```

Strings "comuns"

As strings "comuns" são uma estrutura de dados muito especial que o Delphi trata de maneira única. Primeiro de tudo as strings normais são todas ponteiros. Você não pode simplesmente medir o tamanho delas, apenas o comprimento.

Enquanto as strings de tamanho fixo já têm seu tamanho máximo alocado na Stack, as strings de tamanho variável são alocadas dinamicamente, conforme necessário. Além disso, os oito bytes anteriores ao primeiro caractere, ou seja, offset do -7 ao 0 podem ser divididos entre dois integers de quatro bytes cada um. Um deles armazena a quantidade de caracteres, ou seja, seu comprimento. O outro armazena uma contagem de referências.

Sim, strings são "Reference-Counted" assim como interfaces, e seu mecanismo é controlado nas partes mais internas do Delphi. Isso é necessário para tornar a "cópia" de strings muito mais rápida. Por exemplo, se você tem duas strings, "StringA" e "StringB", sendo que StringA tem o valor 'Clube', quando você fizer a atribuição StringB := StringA não será copiado o inteiro valor da string, cada caractere, porque a string pode ser enorme. É copiado apenas o endereço do primeiro byte, e o contador de referências é incrementado. Se uma das duas strings é modificada, por exemplo com StringB := StringB + 'Delphi', nesse caso o contador de referências é decrementado, a string é verdadeiramente copiada byte a byte e o valor 'Delphi' é concatenado ao final, incrementando o tamanho da mesma. A **Figura 3** ilustra a estrutura interna de uma string.

Se pudéssemos usar um Record para representar uma string ele seria mais ou menos como na **Listagem 9**.

Listagem 9. Representação de como seria a estrutura de uma string

```
TEstruturaString = record
  Referencias: integer;
  Comprimento: integer;
  Caracteres: array[1..MaxInt-8] of Char;
end;
```

Não vamos nos preocupar com o código da **Listagem 9**, porque ele não representa realmente a estrutura das strings, é apenas uma ilustração ou metáfora de como seria. No mundo real a parte de caracteres seria apenas um ponteiro para uma outra estrutura que pudesse armazenar MaxInt-8 caracteres.

Isso porque como já dissemos, as strings não têm um tamanho de dois bilhões de bytes. Esse é seu tamanho máximo, porém o seu tamanho "atual" pode aumentar mediante alocação dinâmica de memória, gerenciada automaticamente pelo Delphi.

O -8 está aqui justamente porque os primeiros 8 bytes são reservados para o contador de referências e o de caracteres, respectivamente.

No código da **Listagem 10** vamos explorar esses contadores de referências e de caracteres e vamos mostrar como uma string já é um ponteiro, tratado de maneira diferente pelo Delphi.

Na **Listagem 10** criamos cinco strings. Três delas tiveram valores constantes, strClube := 'Clube', strDelphi := 'Delphi' e strTitulo := ' – Ponteiros'. Já a StringA é formada dinamicamente, em RunTime, pela concatenação dessas três. Isso é importante porque para uma string com valor constante o contador de referências é sempre -1, que é um valor reservado especial do Delphi. Sempre que uma string é uma constante, vem de uma constante ou de uma constante literal, como no caso strClube := 'Clube', o valor do contador de referências será -1. Se quiser pode fazer o teste com a strClube. Ao concatenar as três strings atribuindo o resultado dinâmico em StringA, esta passa a ficar com 23 caracteres, e o contador de referências passa a ser 1, já que tem apenas uma referência a esse valor.

A cada iteração atualizamos os ponteiros para ter certeza de que estamos apontando para o endereço correto, já que o Delphi pode remanejar os endereços e tamanhos ocupados pelas strings.

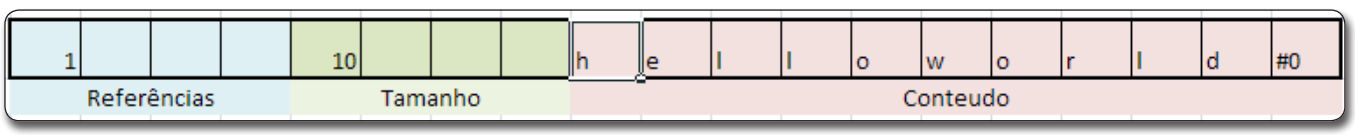


Figura 3. Estrutura interna de uma string

Listagem 10. Obtendo o contador de referências e o de caracteres de uma string

```
procedure TfrmPonteiros.btEx08Click(Sender: TObject);
var
  StringA, StringB, strClube, strDelphi, strTitulo: string;
  Referencias, Comprimento: PInteger;
begin
  strClube := 'Clube';
  strDelphi := 'Delphi';
  strTitulo := ' - Ponteiros';
  StringA := strClube+strDelphi+strTitulo;
  ShowMessage('A string tem o comprimento de ' +
   IntToStr(Length(StringA)) +
   ' Porém sua variável mede apenas '+
     IntToStr(SizeOf(StringA)) +  ' Bytes');
  Referencias := PInteger( integer(StringA)-8);
  Comprimento := PInteger( integer(StringA)-4);
  ShowMessage('A string tem o comprimento de ' +
  IntToStr(Comprimento^) +
   ' E foi referenciada '+ IntToStr(Referencias^) +' vezes');
  Referencias := PInteger( integer(StringA)-8);
  Comprimento := PInteger( integer(StringA)-4);
  //criamos mais uma referência
  StringB := StringA;
  ShowMessage('A string tem o comprimento de ' +
   IntToStr(Comprimento^) +
   ' E foi referenciada '+ IntToStr(Referencias^) +' vezes');
  //modificamos a primeira string, atribuindo o valor da
    segunda e
  //concatenando algo
  StringA := StringB + '-Teste de Referência';
  Referencias := PInteger( integer(StringA)-8);
  Comprimento := PInteger( integer(StringA)-4);
  ShowMessage('A string tem o comprimento de ' +
   IntToStr(Comprimento^) +
   ' E foi referenciada '+ IntToStr(Referencias^) +' vezes');
end;
```

Quando fazemos StringB := StringA as duas passam a ser exatamente idênticas. Por esse motivo, não são copiados os valores dos 23 caracteres para a StringB. A StringB passa a apontar para o mesmo endereço de memória que a string A e seu contador de referências é incrementado para 2, já que agora existem duas strings apontando para o mesmo endereço. Quando modificamos qualquer uma das duas, o contador de referências volta a diminuir e os caracteres são atualizados.

Tudo isso faz com que o Delphi tenha uma excelente performance ao lidar com concatenações de strings, superando em muito as linguagens gerenciadas, mas com um recurso de contagem de referências e alocação dinâmica de strings que você só encontraria em linguagens gerenciadas/interpretadas.

Isso torna o Delphi robusto e rápido, e muito mais fácil de usar do que o C/C++ onde a alocação de espaço em memória para as strings é manual, além das cópias serem feitas através de métodos globais de bibliotecas, e todos os caracteres serem copiados.

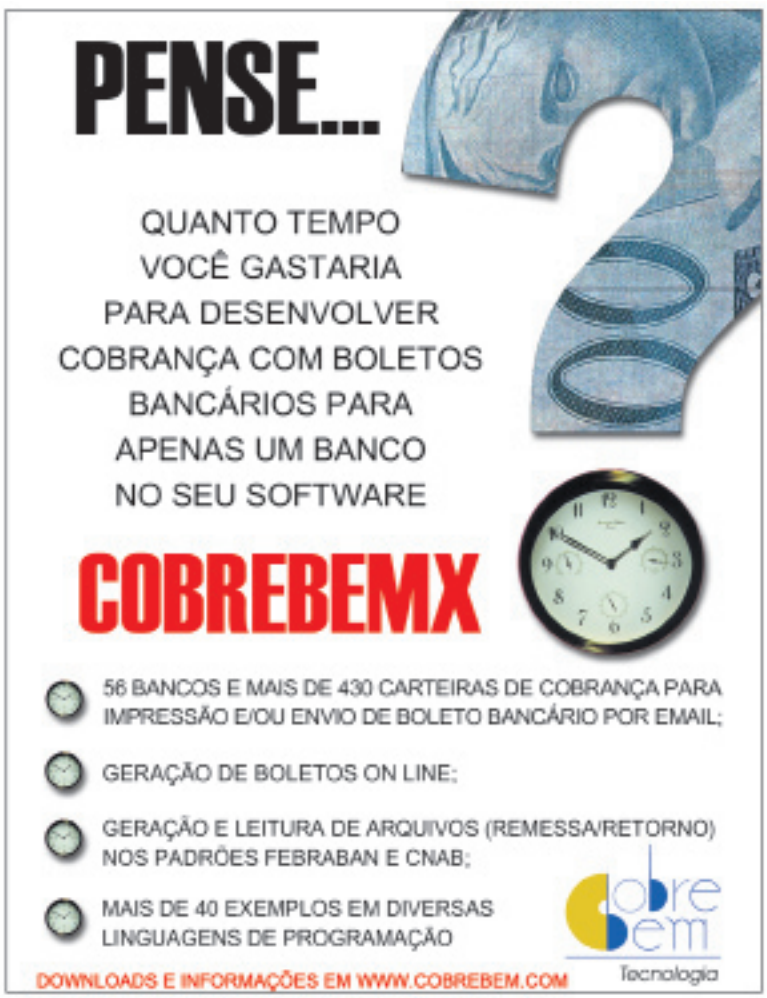
Para se ter uma ideia de como o Delphi é eficiente, comparando com o ambiente .Net, no .Net Framework 1.1, por exemplo, cada concatenação de strings resulta na destruição de uma string e na criação de outra, com o conteúdo completo. Isso resulta num grande overhead quando se estiver criando strings grandes, como arquivos XML/HTML ou grandes textos. Tanto que para textos grandes a Microsoft recomenda usar-se a classe StringBuilder em vez de strings (algo como a TStringList da VCL). A concatenação de strings dentro de um loop, no .Net, seria realmente muito mais lenta que usando o Delphi.

Nota

Se você tentar usar o método SizeOf em uma string comum (de tamanho variável) receberá como resposta apenas o valor 4. Isso ocorre porque uma string é um ponteiro, possuindo apenas 4 bytes.

Ponteiros para métodos

Um caso muito interessante onde podem ser usados ponteiros refere-se aos ponteiros para métodos. Um exemplo prático são as impressoras fiscais. Cada impressora vem com um conjunto de DLL's que deve ser usado para chamar as funcionalidades da impressora. O problema é que você não pode linkar seu software rigidamente, ou estaticamente com todas essas DLL's, muito menos com as DLL's de uma empresa só, se você quiser criar um sistema flexível.



Embora os nomes das funções nas DLL's sejam ligeiramente diferentes, e alguns fabricantes tenham mais funções do que outros, o conjunto básico de funções, imposto pelo Fisco, pode ser encontrado com nomes similares em todas as DLL's.

Você pode sim carregar as DLL's e métodos dinamicamente, dependendo de um arquivo de configuração. Essa é a maneira recomendada de se fazer isso, mas mesmo que você defina estaticamente os cabeçalhos ou interfaces das funções dentro das DLL's você pode criar um conjunto seu de "funções", que são na verdade ponteiros para funções com uma determinada assinatura. Usando esses ponteiros você pode, em uma sessão initialization, dizer quais métodos reais os ponteiros "apontarão".

A **Listagem 11** mostra como isso ficará na prática. Criaremos algumas procedures e functions para substituir outras muito conhecidas do Delphi, e criaremos alguns Types que serão na verdade métodos estáticos globais, ou seja, procedures e functions.

Listagem 11. Ponteiros para métodos

```
private
 { Private declarations }
public
 { Public declarations }
end;
TMostraMensagem = procedure(const  mensagem: string);
var(...)

procedure MensagemPersonalizada(const mensagem: string);
begin
 MessageBox(Application.Handle, pchar(mensagem), 'Mensagem: ',
 MB_OK or
 MB_ICONERROR);
end;
procedure TfrmPonteiros.btEx09Click(Sender: TObject);
var
 ExibeMsg: TMostraMensagem;
 mensagem: string;
begin
 mensagem := 'Hello World';
 //ligando a variável ExibeMsg com a MensagemPersonalizada
 ExibeMsg := MensagemPersonalizada;
 ExibeMsg(mensagem);
 //ligando a variável ExibeMsg com a MensagemPersonalizada
 ExibeMsg := ShowMessage;
 ExibeMsg(mensagem);
end;
```

Você pode estar se perguntando por que não usamos o operador @ antes dos nomes das funções para passar a referência a elas para a variável ExibeMsg. Na verdade isso seria o correto. Se você experimentar fazer esse código em uma versão antiga do Delphi, como o Delphi 3 por exemplo, você será obrigado a usar o operador @. A mesma coisa acontecerá se você estiver usando o Lazarus.

Isso foi introduzido posteriormente no Delphi para "facilitar" esse tipo de atribuições indiretas de métodos. Algumas outras "facilidades" foram introduzidas no Delphi para a manipulação de ponteiros, mas não citaremos essas, pois elas, embora tenham simplificado a sintaxe, afastaram um pouco a linguagem dos padrões do Pascal. Se você desejar fazer um teste, coloque os operadores @ antes dos nomes dos métodos e verá que funcionará normalmente.

O que fizemos na **Listagem 11** foi declarar uma variável do tipo TMostraMensagem, que é definida por:

```
TMostraMensagem = procedure(const  mensagem: string);
```

E atribuímos a esta variável duas procedures com a mesma assinatura, e que fazem praticamente a mesma coisa. Uma é a ShowMessage original do Delphi, da unit Dialogs, e outra que nós mesmos criamos, que internamente usa MessageBox para criar uma mensagem com ícone e título.

Desde a versão 2009, o Delphi tem uma maneira elegante de criar referências a métodos, que são chamados de métodos anônimos. Eles podem ser usados em conjunto com ponteiros para métodos, conforme a **Listagem 12**. As palavras chaves "reference to" também podem ser usadas para criar uma referência (ponteiro) para método. O que vemos adicionalmente é um método anônimo criado dentro do código do evento OnClick do botão.

O código da **Listagem 12** é praticamente idêntico ao código da **Listagem 11**, exceto pelo tipo que declaramos e usamos:

```
TMostraMensagemAnonima = reference to procedure
 (const  mensagem: string)
```

Com esse código criamos um tipo que é um método e podemos tanto usar referências (ponteiros) a métodos como criar métodos InLine, os novos métodos anônimos do Delphi. Com isso também conseguimos mostrar que vários tipos de dados do Delphi são na verdade ponteiros. A última relação que mostraremos será a relação entre ponteiros e objetos.

Ponteiros e objetos: "Hackeando" propriedades read-only

Objetos são ponteiros. Podemos ver isso claramente com o método SizeOf, conforme mostra a **Listagem 13**. Mas os objetos não são ponteiros comuns. São ponteiros para estruturas complexas, com vários níveis de direção. Utilizar o operador ^ em um ponteiro não retornaria nenhum valor.

Conforme visto na **Listagem 13**, um objeto é um ponteiro, e o tamanho de um ponteiro é o mesmo tamanho que os integers: 4 bytes. Porém o tamanho real da instância de um objeto, levando-se em conta as propriedades herdadas de TObject e as propriedades criadas por nós pode ser vista com o método InstanceSize.

Como sabemos, podemos criar propriedades Read-Only, que são definidas conforme operações internas das classes ou durante a construção do objeto. O próprio Delphi tem várias classes com propriedades Read-Only, por exemplo, o objeto Singleton Screen, da classe TScreen contém a propriedade Pixels Per Inch que é Read-Only, definida internamente dentro de sua unit, através da chamada de uma função da API do Windows.

Listagem 12. Métodos anônimos e ponteiros

```
private
 { Private declarations }
public
 { Public declarations }
end;

TMostraMensagem = procedure(const  mensagem: string);
TMostraMensagemAnonima = reference to procedure(const
mensagem: string);

var
(...)

procedure TfrmPonteiros.btEx10Click(Sender: TObject);
var
 ExibeMsg: TMostraMensagemAnonima;
 mensagem: string;
begin
 mensagem := 'Hello World';
 //ligando a variável ExibeMsg com a MensagemPersonalizada
 ExibeMsg := MensagemPersonalizada;
 ExibeMsg(mensagem);
 //ligando a variável ExibeMsg com a MensagemPersonalizada
 ExibeMsg := ShowMessage;
 ExibeMsg(mensagem);
 //criando um código inline para ExibeMsg
 ExibeMsg :=
  procedure(const s: string)
  begin
   ShowMessage('Mensagem Anônima: ' + s);
  end;
 ExibeMsg(mensagem);
end;
```

Listagem 13. Um objeto é um ponteiro

```
procedure TfrmPonteiros.btEx11Click(Sender: TObject);
var
 pessoa: TPessoa;
begin
 pessoa := TPessoa.Create('Ken Masters', 28, 85);
 ShowMessage('O tamanho do objeto, embora tendo uma string,
 um integer e um double é: ' + IntToStr(SizeOf(pessoa)));
 ShowMessage('O tamanho real do objeto, em bytes ocupados no
 heap é: ' +
  IntToStr(pessoa.InstanceSize));
 pessoa.Free;
end;
```

Nota

Na verdade, se pararmos para pensar, não existe nada somente leitura na memória de um computador.

Tentar mudar o valor de uma propriedade somente leitura seria um erro de compilação. Porém, e se conseguíssemos achar um ponteiro para a propriedade, ou pelo menos para os Fields privados internos da classe? Podemos criar um ponteiro com endereço inicial do próprio objeto e ir "somando" números ao endereço desse ponteiro para "vasculhar" onde nosso objeto armazena os valores das propriedades.

Crie uma nova unit chamada uPessoa.pas e implemente nessa unit o código da **Listagem 14**. Faremos uma classe TPessoa simples, com as propriedades somente leitura Nome, Idade e Peso.

Listagem 14. Classe TPessoa

```
unit uPessoa;

interface

type

 TPessoa = class
 private
  FNome: string;
  FIdade: integer;
  FPeso: double;
 public
  property Nome: string read FNome;
  property Idade: integer read FIdade;
  property Peso: double read FPeso;
  constructor Create(aNome: string; aIdade: integer; aPeso: double);
 end;

implementation

{ TPessoa }

constructor TPessoa.Create(aNome: string; aIdade: integer; aPeso:
double);
begin
 FNome := aNome;
 FIdade := aIdade;
 FPeso := aPeso;
end;

end.
```

Uma peculiaridade dessa classe é que todas as propriedades são somente leitura e seus valores são definidos no corpo do construtor. Ao criarmos um objeto desse tipo, uma vez passados os valores, não poderemos mais alterar suas propriedades. O que vamos fazer na **Listagem 15** será criar um ponteiro para cada propriedade, do mesmo tipo que ela, e alterar as propriedades através dos ponteiros. Quebramos assim o conceito de encapsulamento de uma classe do Delphi e expomos seus atributos internos. Não se esqueça de adicionar a nova unit ao uses do frmPonteiros.

Criamos uma instância do objeto TPessoa, com o nome 'Ken Masters', idade 25 e peso 85. Nada de muito extraordinário. Criamos três ponteiros, um para integer, um para string e outro para double. Não precisamos definir esses ponteiros, ou tipos, nem declará-los com o sinal de ^, simplesmente porque o Delphi já define seus tipos da seguinte maneira, na unit System.pas:

```
Type
(...)
PInteger = ^Integer;
PDouble = ^Double;
PString = PUnicodeString;
PUnicodeString = ^UnicodeString;
```

Listagem 15. Desafiando o encapsulamento da OO, com ponteiros

```
procedure TfrmPonteiros.btEx12Click(Sender: TObject);
var
  pessoa: TPessoa;
  pi: PInteger;
  ps: PString;
  pd: PDouble;
begin
  pessoa := TPessoa.Create('Ken Masters', 28, 85);
  pi := @pessoa.Idade;
  ps := @pessoa.Nome;
  pd := @pessoa.Peso;
  ShowMessage('Nome: ' + pessoa.Nome);
  ShowMessage('Idade: ' + inttostr(pessoa.Idade));
  ShowMessage('Peso: ' + FloatToStr(pessoa.Peso));
  ShowMessage('Endereço do objeto: ' + inttostr(integer(pessoa)));
  ShowMessage('Endereço do nome: ' + inttostr(integer(ps)));
  ShowMessage('Endereço da idade: ' + inttostr(integer(pi)));
  ShowMessage('Endereço do peso: ' + inttostr(integer(pd)));
  //Erro: Cannot assign to a read-only property
  {
   pessoa.Nome := 'E. Honda';
   pessoa.Idade := 35;
   pessoa.Peso := 180;
  }
  pi^ := 55;
  ps^ := 'M. Bison';
  pd^ := 90;
  ShowMessage('Nome: ' + pessoa.Nome);
  ShowMessage('Idade: ' + inttostr(pessoa.Idade));
  ShowMessage('Peso: ' + FloatToStr(pessoa.Peso));
  pessoa.Free;
end;
```

Em cada ponteiro armazenamos o endereço de uma propriedade, usando o operador @ no próprio objeto pessoa. Ao exibirmos os endereços de cada ponteiro, inclusive do próprio objeto, podemos reparar que cada propriedade está na memória, imediatamente à distância equivalente ao tamanho de sua predecessora, e que todas elas estão "alinhadas consecutivamente".

Como nossas propriedades são lidas diretamente dos campos, sem passarem por um método de acesso "Get", as propriedades "apontam" diretamente para os Fields privados. Quem faz o controle se a propriedade é somente leitura ou não são as bibliotecas do Delphi. Com os ponteiros que criamos pudemos alterar esses valores somente leitura e alterar o valor das propriedades.

Essa técnica que usamos não é recomendada, pois pode criar efeitos colaterais indesejados em cascata. Fizemos essa experiência apenas como "prova de conceito".

Mesmo assim, se você tiver um daqueles componentes ou bibliotecas de terceiros, antigo, de uma empresa que já fechou, sem fontes e sem suporte e desejar alterar alguma classe ou propriedade, pode criar um descendente que se utilize dessa técnica. Você pode testar isso removendo do projeto a unit uPessoa.pas e renomeando essa unit, mas mantendo no diretório do projeto a uPessoa.dcu compilada. Deixo a cargo do leitor fazer os testes em classes nas quais as propriedades são lidas de um método acessor "Get".

Dica

Você deverá criar um descendente para sua classe, que promova as propriedades somente leitura para published, para poder usar RTTI. Na seção de links você encontrará um post antigo do blog de Hallvard Vassbotn, colunista e editor técnico da revista inglesa The Delphi Magazine e autor de vários livros.

Alocação de memória

Não poderíamos falar sobre ponteiros sem falar sobre a alocação dinâmica de memória. Primeiro, um aspecto teórico, mas importante, é que existem várias áreas de memória: a área Data é onde ficam alocadas nossas variáveis e constantes globais, se é que nós ainda as usamos. A área Stack é onde ficam armazenadas chamadas a métodos, parâmetros dos métodos e variáveis ou arrays internos do escopo de métodos. A área do Heap é onde ficam armazenados os objetos dinâmicos, como objetos. Podemos criar ponteiros para qualquer tipo de variável e alocar blocos de memória no heap, obedecendo as seguintes regras:

1 – Sempre que alocar memória use, dentro de um finally, o método correspondente para desalocar memória, caso contrário criará um vazamento de memória;

2 – Sempre que alocar memória para uma variável usada por um método em uma DLL, caso aloque memória fora da DLL, desaloque também fora da DLL e vice-versa;

3 – Sempre que precisar passar Pchars para DLL's que apenas leem ou usam a string passada, pode-se passar uma constante ou uma string convertida para Pchar, mas sempre que precisar que o conteúdo de uma string seja modificado dentro de uma DLL e devolvido ao Delphi, não use functions, e aloque e desaloque a memória necessária fora da DLL.

Na **Listagem 16** mostramos algumas das diferentes maneiras de se alocar memória.

Na **Listagem 16** podemos ver que memória alocada com new deve ser desalocada com dispose, memória alocada com GetMem deve ser desalocada com FreeMem e memória alocada com GetMemory (que retorna ao endereço do ponteiro) deve ser desalocada com FreeMemory. Existem ainda outras funções da API do Windows, como HeapAlloc e VirtualAlloc para alocar memória no heap e na memória virtual, respectivamente. Esses casos nós não estudaremos nesse artigo.

Cada função para alocar memória tem sua peculiaridade. Por exemplo, new serve somente para alocar um único espaço para um único valor do tipo desejado, no caso do integer, 4 bytes. GetMem pode ser usado para alocar, por exemplo, 20 bytes criando assim um vetor para 5 inteiros. É importante estudar a alocação e destruição de memória para utilizar bibliotecas de terceiros, escritas em outras linguagens, principalmente se forem bibliotecas que controlam dispositivos de hardware específicos.

Conclusão

Embora ponteiros seja um assunto bastante acadêmico, e considerado até mesmo de "baixo nível", este é um assunto

Listagem 16. Alocação dinâmica de memória

```
procedure TfrmPonteiros.btEx13Click(Sender: TObject);
var
 a,b,e: ^integer;
begin
 try
  new(a);
  GetMem(b, sizeof(integer));
  e := GetMemory(sizeof(integer));
  a^ := 5;
  b^ := 10;
  e^ := 100;
  ShowMessage(IntToStr(a^));
  ShowMessage(IntToStr(b^));
  ShowMessage(IntToStr(e^));
 finally
  dispose(a);
  FreeMem(b);
  FreeMemory(e);
 end;
end;
```

essencial caso se deseje interagir com a API do Windows ou com outras API's de outros sistemas. Comunicação com hardware específico, através de bibliotecas que se comunicam com as portas seriais ou paralelas também farão um uso extenso de ponteiros. Na indústria, softwares que controlam máquinas e sensores, termômetros e aplicações em tempo real também fazem um uso extenso de ponteiros. Além disso, se você programa bibliotecas e API's para outros programadores deve ter conhecimento sobre ponteiros e alocação de memória.

Longe de ser um tratado detalhado sobre o assunto, demos apenas uma "pincelada" num assunto tão amplo e que não vai ficar desatualizado enquanto os níveis e camadas mais baixos de nossos sistemas forem escritos em C/C++ e Assembly. Esperamos que esse artigo seja útil para seus ajustes finos e uso de bibliotecas de terceiros. Até a próxima. ●

Links

A Brief History of Strings
http://www.codexterity.com/delphistrings.htm

Hack #1: Write access to a read-only property
http://hallvards.blogspot.com/2004/05/hack-1-write-access-to-read-only.html

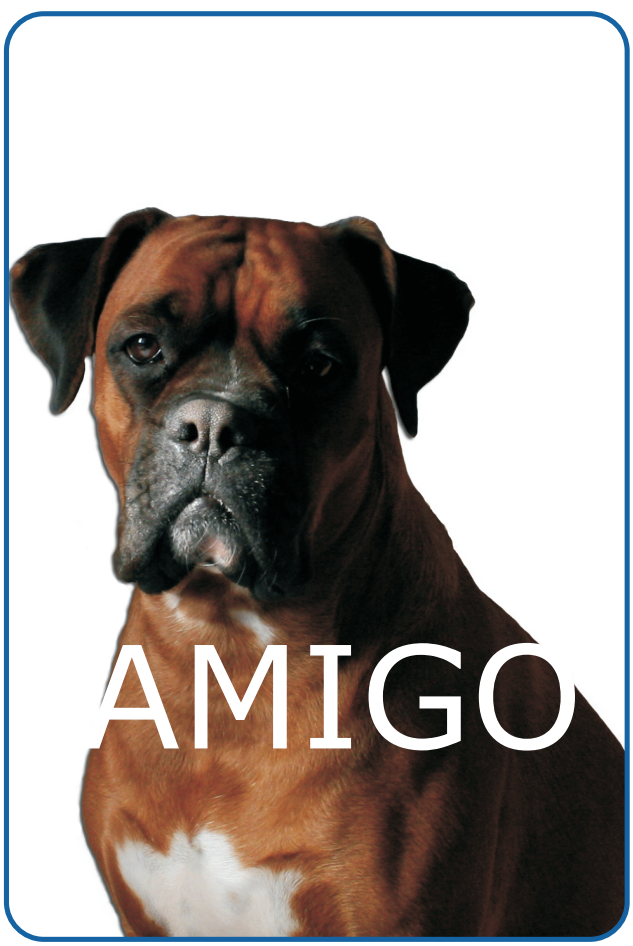
Blogs do autor
http://www.vitorrubio.blogspot.com
http://twitter.com/vitorrubio

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura pode estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

Para mais informações:
www.devmedia.com.br/central

DevMedia GROUP

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Crie um agendador de tarefas OO – Parte 3

Framework de persistência

Rafael Stavarengo
faelsta@gmail.com
http://www.devmedia.com.br/stavarengo
Programador-analista da Cheina Informática. Trabalha com Delphi, JME, JSE e PHP + ZendFramework. Formando em Tecnologia de Análise e Desenvolvimento de Sistemas pela UNIPAR.

Do que trata o artigo
Este artigo ensina como criar frameworks para persistência de dados através da nova RTTI. Aborda conceitos de encapsulamento e polimorfismo, além de mostrar a implementação dos padrões Command, Singleton e Factory com exemplos práticos.

Para que serve
Um programa orientado a objetos precisa converter as informações do banco de dados em objetos antes de manipulá-las. Aprendendo como fazer isto através de frameworks e padrões de projetos você está garantindo uma vantagem em relação à concorrência.

Em que situação o tema é útil
Segundo o que este artigo propõe, você terá um sistema muito fácil de manusear, capaz de atender as necessidades que o dia-a-dia impõe. As técnicas aqui apresentadas serão úteis, por exemplo, para mostrar como usar a nova RTTI no Delphi em um exemplo real, além de fortalecer importantes regras da OO, como separação de responsabilidades e implementação correta de padrões de projeto.

Resumo do DevMan

Se você leu a segunda parte desta série, você aprendeu o quanto é importante criar classes que tenham apenas uma única responsabilidade. Também aprendeu a identificar as responsabilidades de uma classe e como fazer para separá-las. Agora está na hora de criar o banco de dados da aplicação. É claro que o resultado deve ser um código flexível o suficiente para que o SGBD possa ser alterado com pouco esforço.

No último artigo desta série o agendador sofreu grandes modificações. O comportamento do sistema continua o mesmo, pois nenhuma nova função foi adicionada. Porém internamente muita coisa mudou. Em outras palavras, o agendador continua fazendo a mesma coisa, mas de maneira diferente. O principal problema que foi corrigido na segunda versão, era o acúmulo de responsabilidades sobre a classe TForm1, pois esta classe possuía todas a regras de negócios

da aplicação. Após identificar este problema, o que eu fiz, foi separar estas responsabilidades em mais classes e o design do sistema ficou como o apresentado na **Figura 1**.

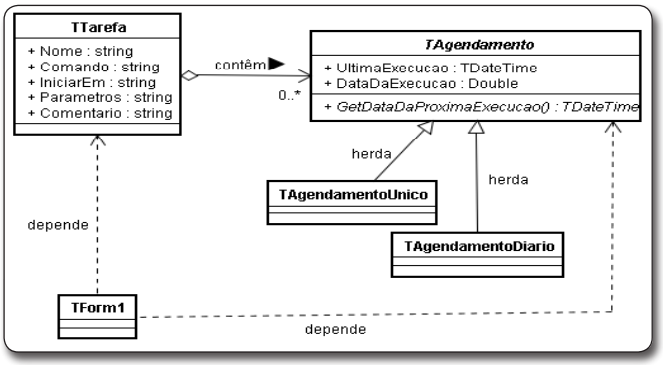


Figura 1. Design atual do sistema

Uma vez que a maioria dos problemas de engenharia foi corrigida, está na hora de implementar uma das funções mais importantes: salvar as informações no banco de dados. Até agora o aplicativo não persiste as informações, ao invés disto, cada tarefa é adicionada a uma variável do tipo TList e é claro que quando o sistema é fechado esta lista é perdida. Agora que o sistema já está mais maduro é possível implementar a persistência dos objetos.

O objetivo da alteração

É preciso retirar da classe TForm1 toda a responsabilidade referente ao ato de salvar informações. Como você pode ver na **Listagem 1**, o código para salvar uma tarefa é muito simples, basta adicionar a nova tarefa na variável utilizada como banco de dados temporário da aplicação – veja linha 11 da **Listagem 1**. Normalmente é necessário mais código para salvar um registro como, por exemplo, abrir uma conexão, criar a tabela caso não exista, verificar se já existe um registro com o mesmo nome e somente depois salvar a tarefa. O que atualmente no agendador é representado apenas por uma linha, na vida real seriam várias linhas de códigos. O objetivo é separar toda essa lógica em classes que sejam responsáveis exclusivamente por persistir os objetos.

Listagem 1. Código para criar e adicionar uma nova tarefa

```
1   procedure TForm1.SalvarTarefaClick(Sender: Tobject);
2    var
3    Tarefa: Ttarefa;
4   begin
5    Tarefa := Ttarefa.Create;
6    Tarefa.Comando := Comando.Text;
7    Tarefa.Nome := Nome.Text;
8    Tarefa.Parametros := Parametros.Text;
9    Tarefa.IniciarEm := IniciarEm.Text;
10   Tarefa.Comentario := Comentarios.Text;

11   Lista.Add(Tarefa);

12   MostrarTarefas;
13  end;
```

Quando eu digo persistir os objetos, me refiro à ação de salvar as informações de todas as propriedades de um objeto no banco de dados e quando necessário for, buscar e colocar essas informações de volta em um objeto do mesmo tipo. Fazer isto de forma manual é possível, mas muito cansativo e o resultado não será bom, pois cada objeto possui diferentes propriedades. Por este motivo é óbvio que isso deve ser feito de modo automático.

Isto cria um grande problema: como obter as informações das propriedades de um objeto sem saber quais são as propriedades dele? Como alterar o valor da propriedade de um objeto se eu não sei nem de qual objeto eu estou falando?

A solução para isso é RTTI (Run-time Type Information). Você provavelmente já sabe que o Delphi 2010 trouxe uma nova RTTI cheia de facilidades e novos recursos (veja sessão Links). É com ela que os problemas apresentados no último parágrafo serão resolvidos. É claro que você também consegue o mesmo resultado com a antiga RTTI do Delphi, porém neste artigo você vai aprender a montar um mini-framework de persistência usando a nova RTTI.

Antes de inserir um registro

Para inserir um registro em uma tabela, primeiro você precisa que a tabela exista, portanto, isto significa que será necessário que o framework crie a tabela automaticamente caso ela ainda não exista.

No caso do agendador as informações serão persistidas em XML, mas como você sabe este é apenas um detalhe. A única diferença para programas que usam banco de dados relacionais é que você terá que digitar um pouco de código SQL em seu framework – ou talvez não, depende do nível de abstração que você for usar.

Veja na **Figura 2** como o framework é simples. A classe TBean mostrada representa as classes que serão persistidas pelo framework. Mais do que isso, o framework só vai persistir objetos das classes que herdem a classe TBean, que é abstrata. Ela é abstrata justamente porque a ideia é que você a estenda nas classes que serão persistidas pelo framework. No caso do agendador serão persistidas as classes TTarefa e todas as filhas de TAgendamento.

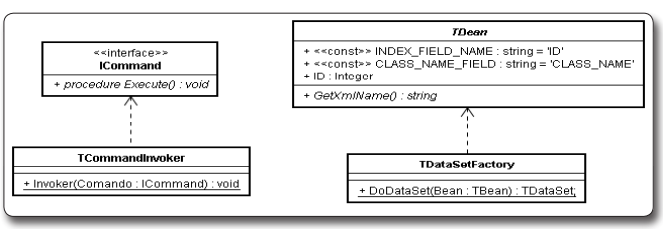


Figura 2. Design do framework

A classe TBean possui algumas características importantes. Em primeiro lugar vou destacar as duas constantes desta classe, são elas: INDEX_FIELD_NAME e CLASS_NAME_FIELD. A constante INDEX_FIELD_NAME representa o nome do campo que será usado como chave primária da tabela. Esta constante será muito importante para o framework na hora

de criar a tabela. A constante CLASS_NAME_FIELD é o nome do campo onde será armazenado o nome da classe que uma determinada linha da tabela representa. Este campo será importante para situações como a da classe TAgendamento. No caso desta classe, não é a classe TAgendamento que será persistida, mas sim as suas subclasses. Isto significa que cada linha da tabela de agendamentos poderá estar representando um tipo de agendamento diferente, ou seja, uma subclasse de agendamento diferente. Logo você vai entender melhor e ver isto funcionando de verdade.

Ainda falando sobre a classe TBean, veja que ela possui um atributo ID. Uma vez que todas as tabelas possuem uma chave primária, a classe TBean já faz a declaração deste atributo para você. O atributo ID da classe TBean está diretamente relacionado com o valor da constante INDEX_FIELD_NAME. O nome deste atributo deve ser igual ao valor da constante INDEX_FIELD_NAME, pois esta constante identifica o nome do atributo usado como chave primária da tabela. Por fim, temos a declaração do método abstrato GetXmlName. Este método deve retornar o nome do arquivo XML usado para persistir as informações do objeto, em outras palavras, é o objeto TBean que sabe o nome do arquivo onde suas informações serão persistidas.

Uma fábrica dentro do framework

Existe uma fábrica dentro do framework e o nome dela é TDataSetFactory (**Listagem 2**). Esta classe tem o objetivo de fabricar os objetos TDataSet devidamente configurados para estabelecer conexão com o arquivo onde as informações serão persistidas. Observe que o único método disponível nesta classe é o DoDataSet. Quando invocado este método vai criar e configurar um objeto TDataSet para que ele se conecte com o arquivo correspondente ao objeto Bean passado como parâmetro para o método. É claro que se o arquivo para o qual o TDataSet vai se conectar não existir a fábrica deve criá-lo.

Internamente esta fábrica faz duas coisas que merecem destaques. Primeiro ela verifica se o arquivo XML, referente ao objeto Bean recebido como parâmetro, existe. Se não existir a fábrica o cria de acordo com as propriedades do objeto Bean. O segundo fato importante é que a aplicação usa arquivos XML justamente porque esta fábrica cria objetos TDataSet configurados para conectar em arquivos XML. Se futuramente for necessário trocar o tipo de banco de dados do agendador para Firebird, por exemplo, será muito simples, pois o maior trabalho – talvez o único – será criar uma nova fábrica que crie objetos TDataSet configurados para conectar com uma tabela específica do banco Firebird. Criada a nova fábrica, o agendador não vai nem sentir a diferença de estar se usando banco de dados.

TDataSetFactory fabrica objetos TDataSet porque quase todos, se não todos, os componentes de conexão utilizam a classe TDataSet. Portanto trocar o SGBD, ou o componente de conexão se torna muito simples, pois a fábrica retorna o objeto mais genérico entre eles.

ICommand e TCommandInvoker

Este framework traz consigo a implementação do padrão Command. Ou pelo menos traz parte da implementação, pois ainda estão faltando as classes de comando que serão criadas mais adiante. Como você deve saber, as classes de comando podem executar qualquer coisa e neste artigo existirão duas classes de comando: uma classe para salvar, persistir um objeto no XML e outra classe de comando para buscar as tarefas que já foram salvas.

Codificando o framework

Primeiro você vai conhecer a classe TDataSetFactory. Lembre-se que esta é a classe do framework que vai criar automaticamente as tabelas do banco e o TDataSet configurado para conectar com a tabela criada. Lembre-se também que é esta classe quem define o tipo de banco de dados que está sendo usado, veja **Listagem 2**.

Listagem 2. Interface da classe TDataSetFactory

```
1   unit DataSetFactory;

2   interface

3   uses Db, DBClient, Bean, Rtti;

4   type
5    TDataSetFactory = class
6    private
7     function CreateDataSet(Bean: TBean; ArquivoXml: string):
       TClientDataSet;
8     procedure AdicionarCampo(DataSet: TDataSet; Atributo:
       TRttiField);
9    protected
10    constructor Create; reintroduce;
11    class function GetFactory: TDataSetFactory;
12   public
13    class function DoDataSet(Bean: TBean): TDataSet;
14   end;

15  implementation

16  uses SysUtils, TypInfo, Classes, Forms;

17   var
18    Factory: TDataSetFactory;
```

A **Listagem 2** mostra a interface da classe TDataSetFactory. Como você já sabia esta classe possui apenas um método público: DoDataSet. Este método recebe como parâmetro um objeto TBean e retorna um TDataSet configurado para conectar com o arquivo XML do objeto TBean passado como parâmetro. Na verdade quem faz isso é o método CreateDataSet que foi declarado como privado na linha 7. Este método é que vai criar, caso não exista, o arquivo XML e depois criar o TDataSet propriamente dito. O método AdicionarCampo (linha 8) faz exatamente o que seu nome diz. Ele cria um novo campo no objeto TDataSet de acordo com a propriedade do objeto. Na linha 10 está o construtor e na linha 11 está o método GetFactory. O construtor está declarado como protegido, pois não é necessário que exista mais de uma fábrica no sistema.

Protegendo o construtor torna-se impossível que exista mais de um objeto fábrica, já que ele não estará acessível a outras classes. O método GetFactory é o responsável por criar a única instância da fábrica que vai existir. Este método é a implementação do padrão Singleton. Se você não conhece este padrão, logo você verá como ele funciona.

Nota

Você notou a palavra class no início das linhas 11 e 13 da **Listagem 2**? Métodos com esta assinatura são chamados de métodos estáticos ou métodos de classe. Este tipo de método pode ser chamado diretamente da classe sem a necessidade de criar um objeto para chamá-lo.

Agora temos o mais interessante do framework: desmontar e ler um objeto a fim de identificar quais são os atributos e então criar um arquivo XML com um campo para cada atributo existente. Isto pode ser visto na **Listagem 3**.

Tudo começa com o método DoDataSet – linha 1. Lembra do método GetXmlName da classe TBean? É neste momento que ele é usado, pois o método DoDataSet invoca GetXmlName para montar o nome completo do arquivo XML. Você não viu ainda, mas o método GetXmlName do objeto TBean usado na linha 6 vai retornar algo assim: "Agendamento.xml".

Listagem 3. Fabricando um TDataSet

```
1  class function TDataSetFactory.DoDataSet(Bean: TBean): TDataSet;
2   var
3    Fabrica: TDataSetFactory;
4    ArquivoXml: string;
5  begin
6    ArquivoXml := ExtractFilePath(Application.ExeName) + Bean.
     GetXmlName;
7    Fabrica := GetFactory;
8    Result := Fabrica.CreateDataSet(Bean, ArquivoXml);
9  end;

10 function TDataSetFactory.CreateDataSet(Bean: TBean;
   ArquivoXml: string)
11  : TClientDataSet;
12 var
13  RttiType: TRttiType;
14  Campos: TArray<TRttiField>;
15  Campo: TRttiField;
16 begin
17  Result := TClientDataSet.Create(nil);
18  Result.FileName := ArquivoXml;

19  if not FileExists(ArquivoXml) then
20  begin
21   RttiType := TRttiContext.Create.GetType(Bean.ClassInfo);
22   Campos := RttiType.GetFields;
23   for Campo in Campos do
24    AdicionarCampo(Result, Campo);

25   with Result.FieldDefs.AddFieldDef do
26   begin
27    Name := TBean.CLASS_NAME_FIELD;
28    DataType := ftString;
29    Size := 1000;
30   end;

31   Result.CreateDataSet;
32   Result.SaveToFile(ArquivoXml, dfXML);
33  end;

34  Result.IndexFieldNames := TBean.INDEX_FIELD_NAME;
35  Result.Close;
36 end;
```

Na linha 8 o método CreateDataSet é invocado. Na linha 17 eu crio um objeto TClientDataSet. É este objeto que será retornado no final do método. Lembra-se do que eu falei sobre o TDataSet ser o objeto mais genérico dos objetos de conexão? Repare que internamente a fábrica está criando um objeto TClientDataSet, mas o método CreateDataSet retorna um TDataSet. Isto é o polimorfismo em ação. Da mesma forma você poderia criar um objeto do tipo TIBQuery, pois ele também é um TDataSet. Assim você pode trocar a fábrica que está sendo usada por outra que cria objetos de conexão com tabelas do Firebird sem que o agendador note o que foi mudado, pois ele continuará trabalhando com objetos TDataSet sem saber se o objeto é na verdade um TClientDataSet, TIBQuery ou outro qualquer.

Note, na linha 17, que eu crio o TClientDataSet dentro da própria variável Result. Em seguida eu defino o nome do arquivo que o TClientDataSet vai se conectar. Na linha 19, se o arquivo XML ainda não existir é necessário criá-lo. Para isto, na linha 21 eu pego as informações RTTI do objeto TBean passado para a fábrica. A variável RttiType foi declarada na linha 13 e ela é do tipo TRttiType. Este tipo é a base para a manipulação das informações RTTI de um objeto. Através dele você tem acesso à estrutura de uma classe em tempo de execução. Para conseguir uma instância de TRttiType de qualquer classe é necessário usar o método GetType do record TRttiContext, como você poder ver na linha 21. Uma vez que estou com o objeto do tipo TRttiContext em mãos fica fácil pegar uma lista de todos os atributos do objeto TBean para criar campos no TDataSet. Basta invocar o método GetFields, veja linha 22. Este método retorna um array de TRttiField. Cada item do array retornado será um atributo do objeto TBean. Lembre-se que TBean pode ser qualquer objeto que você queira persistir em um banco de dados. No caso do agendador são os objetos da classe TTarefa e todos os objetos das subclasses de TAgendamento.

Na linha 23 é feita uma iteração com o array de atributos retornado pelo método GetFields e para cada atributo no array um campo é adicionado ao TDataSet. Logo você vai ver a implementação do método AdicionarCampo, invocado na linha 24.

Nota do DevMan

O padrão Singleton cria um ponto comum de acesso para um objeto de uma classe, de maneira que se tenha apenas uma única instância desse objeto. Ao esconder o construtor da classe se torna impossível criar um objeto da maneira convencional, que seria TNomeDaClasse.Create. Ao invés disto uma classe singleton deve disponibilizar um método de classe, normalmente chamando de GetInstance – que neste exemplo é chamado de GetFactory – para obter o único objeto da classe que é usado por toda a aplicação. Uma das vantagens da utilização deste padrão é o baixo uso de recursos de hardware, pois garante que não existirá mais de um objeto para uma mesma classe, poupando o uso de memória.

Entre as linhas 25 e 30 mais um campo é criado no TDataSet. O campo criado neste trecho de código é aquele campo que vai guardar o nome da classe que as informações de uma determinada linha dentro da tabela XML representa. Por fim, na linha 31 o arquivo XML é criado, porém apenas em memória. Na linha 32 o arquivo XML é salvo no HD. A variável ArquivoXml usada nesta linha possui o caminho completo do arquivo XML a ser usado, conforme você viu na linha 6 da **Listagem 3**. Na linha 34 é definido que o campo de ID é a chave primária do TDataSet criado. Na linha 35 a conexão com o arquivo XML é fechada. Neste momento o TDataSet já está configurado para conectar com o arquivo XML.

Adicionando um campo ao TDataSet

Como você viu na linha 24 da **Listagem 3**, para cada atributo dentro do array de atributos do objeto TBean é feita uma chamada ao método AdicionarCampo. Este método vai identificar o tipo do atributo – string, Integer etc. – e criar um campo dentro do TDataSet configurado para guardar o tipo correto de informação. Veja **Listagem 4**.

Listagem 4. Adicionando um campo ao TDataSet

```
1   procedure TDataSetFactory.AdicionarCampo
    (DataSet: TDataSet; Atributo: TRttiField);
2    var
3    NovoCampo: TFieldDef;
4   begin
5    NovoCampo := DataSet.FieldDefs.AddFieldDef;
6    case Atributo.FieldType.TypeKind of
7     tkLString, tkString, tkEnumeration, tkUString:
8      begin
9       NovoCampo.DataType := ftString;
10      NovoCampo.Size := 1000;
11     end;

12    tkInteger, tkChar, tkSet, tkWChar:
13     begin
14      if Atributo.Name = TBean.INDEX_FIELD_NAME then
15       NovoCampo.DataType := ftAutoInc
16      else
17       NovoCampo.DataType := ftInteger;
18     end;

19    tkFloat:
20     NovoCampo.DataType := ftFloat;

21  else
22   begin
23    DataSet.FieldDefs.Delete(NovoCampo.Index);
24    Exit;
25   end;
26  end;
27  NovoCampo.Name := Atributo.Name;
28 end;
```

O método AdicionarCampo exibido na **Listagem 4** recebe dois parâmetros: DataSet do tipo TDataSet e Campo do tipo TRttiField. O primeiro parâmetro é o objeto TDataSet onde o campo criado será adicionado. O segundo parâmetro é um objeto que representa a estrutura de um determinado atributo de uma classe. O tipo TRttiField representa a estrutura de um atributo. Por exemplo: a classe TAgendamento possui o atributo Comando, como você viu na **Figura 1**. Para saber em tempo de execução qual o tipo de informação que este atributo guarda é necessário um objeto TRttiField. Agora que você entendeu mais sobre este objeto da nova RTTI do Delphi, está na hora de entender como um campo é adicionado ao TDataSet.

Primeiro um novo campo é criado dentro do TDataSet como você pode ver na linha 5. Depois na linha 6 começa uma instrução "case" que vai testar o tipo de informação que o atributo do objeto TBean armazena. A instrução Atributo.FieldType.TypeKind na linha 6 vai retornar o tipo de informação que o atributo do objeto armazena.

Se o atributo armazenar informações do tipo tkLString, tkString, tkEnumeration ou tkUString o novo campo do TDataSet é configurado para armazenar informações do tipo string, como você pode ver entre as linhas 7 e 11. Se o atributo armazenar informações do tipo tkFloat o campo é configurado para guardar informações do tipo Float, como você pode ver nas linhas 19 e 20.

Se o atributo armazenar informações do tipo tkInteger, tkChar, tkSet ou tkWChar, o novo campo é configurado para armazenar informações do tipo Integer, porém existe uma peculiaridade para atributos com este tipo. Um campo do tipo Integer pode ser na verdade o campo chave primária da tabela e se for o caso, o campo deve ser autoincremento. É isto que acontece entre as linhas 12 e 18. Se for um atributo de números inteiros primeiro é verificado se não é o atributo que deve ser a chave primária. Se for, configura o novo campo do TDataSet para ser autoincremento, caso contrário configura o campo para armazenar informações do tipo Integer.

Se o atributo não for de nenhum dos tipos citados anteriormente, o novo campo criado na linha 5 é excluído do TDataSet e em seguida a execução do método é interrompida. Por fim, depois do novo campo do TDataSet estar devidamente configurado, falta apenas dar o nome deste campo. O nome do campo será o mesmo nome do atributo do objeto TBean. Veja linha 27.

Ao final da execução do método AdicionarCampo apresentado na **Listagem 4**, um novo campo vai existir dentro do objeto TDataSet. Este novo campo só não vai existir se o atributo do objeto for de um tipo não reconhecido.

Uma fábrica sem igual

Os métodos mais interessantes da fábrica TDataSetFactory já foram expostos para você. Mas ainda faltam dois. São os dois métodos necessários para a implementação do padrão Singleton. Veja **Listagem 5**.

O método GetFactory apresentado na linha 1 da **Listagem 5** é o método que garante que apenas uma única instância da classe TDataSetFactory vai existir. Veja que este método usa a variável Factory para guardar a única instância da fábrica. A declaração desta variável está na linha 18 da **Listagem 2**. Na linha 7 está a implementação do construtor. Ele foi declarado como protegido na linha 10 da **Listagem 2**. Repare que ele não executa nenhuma lógica extra. Ele foi redeclarado na classe TDataSetFactory apenas para poder ficar protegido contra o acesso das outras classes e assim contemplar a implementação do padrão Singleton.

Listagem 5. Implementação do padrão Singleton

```
1  class function TDataSetFactory.GetFactory: TDataSetFactory;
2  begin
3   if not Assigned(Factory) then
4     Factory := TDataSetFactory.Create;
5    Result := Factory;
6  end;

7  constructor TDataSetFactory.Create;
8  begin
9   inherited Create;
10 end;
```

A classe abstrata TBean

Até agora você viu que a classe TBean é usada pela classe TDataSetFactory praticamente durante toda a produção de um TDataSet. Isto acontece porque a fábrica pode instanciar TDataSets para qualquer objeto, desde que este objeto estenda a classe TBean. Assim pode-se dizer que a fábrica está usando polimorfismo, também para poder fabricar TDataSets que vão armazenar informações de qualquer objeto passado para a fábrica. Veja o código da classe TBean na **Listagem 6**.

Listagem 6. A classe TBean

```
1  unit Bean;

2  Interface

3  uses Classes;

4  Type
5    TBean = class abstract (TPersistent)
6    public
7        ID: Integer;
8        const INDEX_FIELD_NAME = 'ID';
9        const CLASS_NAME_FIELD = 'CLASS_NAME';
10   function GetXmlName: string; virtual; abstract;
11   end;

12 Implementation

13 end.
```

Veja a **Listagem 6**. Ela apresenta a classe TBean, mas seja sincero: não parece que você já conhecia esta classe? Note que não existe nada a mais na **Listagem 6** do que o diagrama da classe TBean apresentado na **Figura 2**. Lembre-se: sempre que você quiser passar um objeto para que a fábrica crie o TDataSet para este objeto, ele deve herdar as características da classe TBean. Isto é o que você precisa saber sobre a classe TBean. Além é claro de saber que o método GetXmlName vai retornar o nome do arquivo XML onde as informações do objeto que herdou as características da classe TBean vão armazenar as informações. Agora, você já está pronto para ver como tudo isto funciona.

Colocando a fábrica para fabricar

Com a fábrica codificada e após entender como a classe TBean deve ser usada, está na hora de usar esta fábrica no agendador de tarefas. Mas antes de começar a modificar o agendador para usar a fábrica de TDataSet é necessário fazer um teste para garantir que a fábrica está funcionando corretamente.

No Delphi 2010 crie um novo projeto e feche o formulário inicial que o Delphi sugere sem salvar. Depois disto salve o projeto com o nome de TesteDaFabrica.dproj. Agora crie uma nova unit através do menu File > New > Unit – Delphi. Salve esta nova unit com o nome de DataSetFactory.pas. Nesta unit você deve digitar o código das **Listagens 2, 3, 4** e **5** respectivamente. Neste momento se você tentar compilar o projeto um erro vai ser gerado porque a classe TBean não existe. Por tanto, crie mais uma nova unit e salve ela com o nome de Bean.pas. Nesta unit você deve digitar o código da **Listagem 6**. Agora se você compilar o projeto nenhum erro será gerado.

Chegou a hora de criar uma classe para que a fábrica possa criar o arquivo XML onde as informações desta classe serão salvas. Crie uma nova unit e salve esta unit com o nome de PessoaTest.pas. Digite o código da **Listagem 7** nesta nova unit.

Listagem 7. A classe TPessoaTest

```
1  unit PessoaTest;

2  interface

3  type
4   TPessoaTest = class
5   public
6     Nome: string;
7     Idade: Integer;
8   end;

9  implementation

10 end.
```

A classe TPessoaTest é uma simples classe que vamos usar de teste para verificar se a fábrica TDataSetFactory está gerando um arquivo XML com as informações corretas de acordo com os atributos da classe que lhe é passado. Neste caso quando for pedido para a fábrica criar o TDataSet da classe TPessoaTest, um arquivo XML com os campos Nome e Idade deve ser criado. Vamos ao teste. Abra o arquivo DPR deste novo projeto de teste através do menu Project > View Source. Em seguida digite o código da **Listagem 8** no arquivo DPR.

Listagem 8. Executando um teste

```
1  {$R *.res}

2   var
3  Pessoa: TPessoaTest;
4  begin
5    Application.Initialize;

6    Pessoa := TPessoaTest.Create;
7    TDataSetFactory.DoDataSet(Pessoa);

8    Application.Run;
9  end.
```

A **Listagem 8** cria um objeto de pessoa e pede para a fábrica criar um TDataSet que possa ser usado para persistir as informações que a classe TPessoaTest possui: neste caso é o nome e a idade da pessoa. Tente compilar este projeto. Você não vai conseguir. Ao compilar aparece um erro na linha 7. Este erro diz que a classe TPessoaTest não é compatível com a classe TBean. Isto acontece porque o método DoDataSet invocado na linha 7 recebe como parâmetro um objeto TBean e se você notar na linha 4 da **Listagem 7**, TPessoaTest não é um objeto TBean. Para corrigir este problema você deve fazer com que a classe TPessoaTest herde as características de TBean. Para isto substitua o que está escrito na linha 4 da **Listagem 7** por:

```
TPessoaTest = class(TBean)
```

É claro que para isto funcionar você deve adicionar a seguinte instrução entre as linhas 2 e 3 da **Listagem 7**: "uses Bean;". Feito isto, você pode compilar e executar o projeto novamente. Não funcionou não é mesmo? Gerou uma exceção em tempo de execução dizendo alguma coisa relacionada a "erro abstrato". Sim, isto era esperado. A classe TPessoaTest agora herda de TBean. Por sua vez TBean possui um método abstrato e este método precisa ser implementado, caso contrário este erro vai continuar aparecendo. Veja na **Listagem 9** a implementação do método GetXmlName na classe TPessoaTest. Considere que o código da **Listagem 9** está entre as linhas 9 e 10 da **Listagem 7**.

Listagem 9. Implementando o método GetXmlName

```
1  {TPessoaTest}

2  function TPessoaTest.GetXmlName: string;
3  begin
4    Result := 'PessoaTest.xml';
5  end;
```

Depois de colocar o código entre as linhas 9 e 10 da **Listagem 7**, você precisa escrever a seguinte instrução entre as linhas 7 e 8 da **Listagem 7**:

```
function GetXmlName: string; override;
```

Pronto. Agora vai funcionar, eu garanto. Veja o código completo da classe TPessoaTest na **Listagem 10**.

Após executar o programa, você terá a impressão de que nada aconteceu. Pois ele executa rapidamente e logo em seguida fecha sozinho. Mas este é o esperado. Abra a pasta onde você salvou o projeto e veja que existe um arquivo chamado PessoaTest.xml. Abra este arquivo no seu editor de XML preferido e você verá que ele possui todos os campos para persistir o nome e a idade da classe TPessoaTest, além de já possuir um campo chamado ID que vai ser usado de chave primária e um campo chamado CLASS_NAME onde será armazenado o nome da classe que guardou informações em uma determinada linha desta tabela. Veja **Figura 3**.

Listagem 10. A classe TPessoaTest pronta para ser usada

```
1   unit PessoaTest;

2   interface

3   uses Bean;

4   type
5    TPessoaTest = class(TBean)
6    public
7      Nome: string;
8      Idade: Integer;
9      function GetXmlName: string; override;
10   end;

11  implementation

12  {TPessoaTest}

13  function TPessoaTest.GetXmlName: string;
14  begin
15    Result := 'PessoaTest.xml';
16  end;

17  end.
```

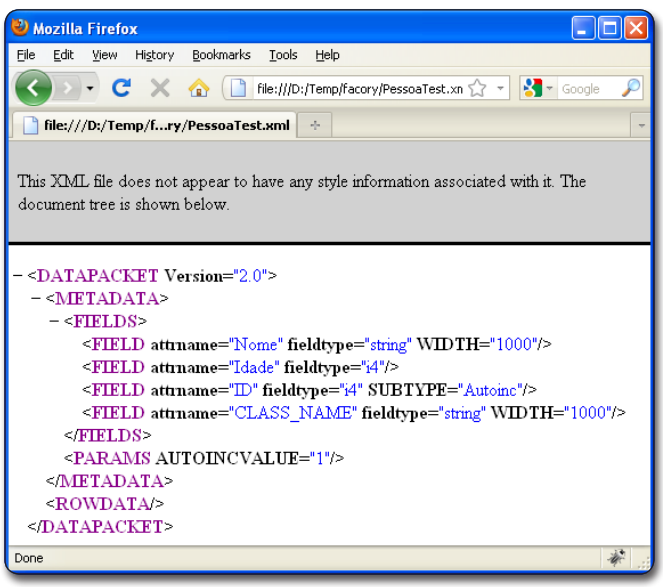


Figura 3. O arquivo XML gerado pela fábrica

Um teste mais completo

Agora está na hora de fazer um teste um pouco mais completo. Que tal inserir um registro neste arquivo XML. Veja na **Listagem 11** como fica o código do arquivo DPR para executar este teste.

Após executar o teste apresentado na **Listagem 11** o arquivo XML deve possuir um novo registro, de acordo com o apresentado na **Figura 4**. A única coisa que vou destacar nesta última listagem é a instrução da linha 20, onde o campo CLASS_NAME do TDataSet recebe o nome da classe que está salvando as informações.

Você viu que o arquivo XML está sendo criado automaticamente, porém inserir um novo registro não foi nem um pouco automático. É aí que entram as classes de comandos

citadas no tópico ICommand e TCommandInvoker. A ideia é que existam classes de comandos para automatizar o processo de salvar e localizar um registro dentro do TDataSet. Antes de criar as classes de comando vou mostrar o código da classe TCommandInvoker e da interface ICommand. Veja as **Listagem 12** e **13**.

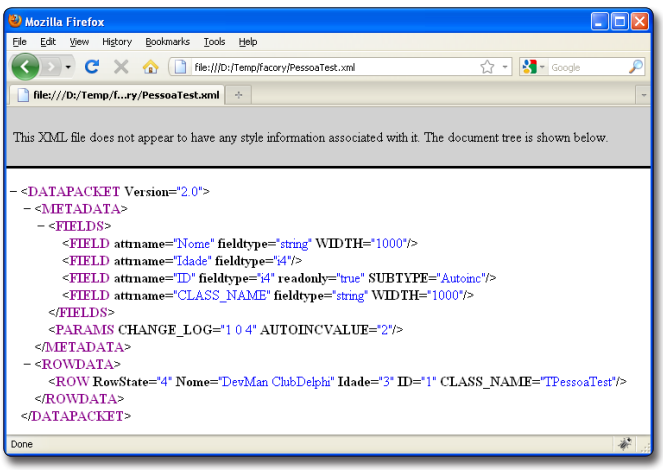


Figura 4. Um novo registro no arquivo XML

Nas **Listagens 12** e **13** está o código da classe TCommandInvoker e da interface ICommand. Não há nada especial que mereça destaque nestas listagens. É claro que se você não conhece o funcionamento do padrão Command talvez esteja se sentindo um pouco confuso. Neste caso recomendo que leia o artigo Padrão Command, indicado na sessão Links. Veja agora o código da **Listagem 14**.

A **Listagem 14** apresenta uma parte da classe de comando TSalvarCommand. Esta classe será capaz de persistir qualquer objeto no banco de dados, indiferentemente do banco de dados utilizado, pois ela vai trabalhar com base no objeto TDataSet retornado pela fábrica e como você já viu, o TDataSet pode ser na verdade uma variedade de outros componentes de comunicação com o banco de dados. Lembre-se também que os objetos precisam herdar da classe abstrata TBean para poderem ser persistidos.

Com estas considerações em mente, veja que o construtor da classe de comando TSalvarCommand, declarado na linha 9 da **Listagem 14**, recebe um objeto do tipo TBean. Isto significa que para criar um objeto da classe TSalvarCommand é necessário dar em troca um objeto da classe TBean. Este é o objeto que será persistido pela classe de comando. Entre as linhas 15 e 19 está a implementação do construtor. Veja na linha 18 que o construtor armazena a referência ao objeto TBean passada como parâmetro no atributo também chamado de Bean. Este atributo está declarado na linha 7. Veja a declaração do método Execute na linha 10. É este método que vai persistir as informações do objeto TBean. Um último detalhe, veja na linha 5 que a classe TSalvarCommand implementa a interface ICommand.

Vale lembrar: com a classe de comando TSalvarCommand o processo de persistir um objeto no banco de dados vai ser automatizado. Em outras palavras todo o código apresentado

Listagem 11. Inserindo um registro

```
1  program TesteDaFabrica;

2  uses
3   Forms,
4   DataSetFactory in 'DataSetFactory.pas',
5   Bean in 'Bean.pas',
6   PessoaTest in 'PessoaTest.pas',
7   DB;
8  {$R *.res}

9   var
10  Pessoa: TPessoaTest;
11  Dataset: TDataSet;

12 begin
13  Application.Initialize;

14  Pessoa := TPessoaTest.Create;
15  Dataset := TDataSetFactory.DoDataSet(Pessoa);

16  Dataset.Open;
17  Dataset.Insert;
18  Dataset.FieldByName('Nome').AsString := 'DevMan ClubeDelphi';
19  Dataset.FieldByName('Idade').AsInteger := 3;
20  Dataset.FieldByName(Pessoa.CLASS_NAME_FIELD).AsString :=
     Pessoa.ClassName;
21  Dataset.Post;
22  Dataset.Close;

23  Application.Run;

24 end.
```

Listagem 12. Interface de comandos

```
1  unit Command;

2  interface

3  type
4   ICommand = interface
5    procedure Execute;
6   end;

7  implementation

8  end.
```

Listagem 13. O invocador de comandos

```
1  unit CommandInvoker;

2  interface

3  uses Command;

4  type
5   TCommandInvoker = class
6   public
7    class procedure Invoker(Comando: ICommand);
8   end;

9  implementation

10 { TCommandInvoker }

11 class procedure TCommandInvoker.Invoker(Comando: ICommand);
12 begin
13  Comando.Execute;
14 end;

15 end.
```

entre as linhas 15 e 22 da **Listagem 11** será automatizado pela classe TSalvarCommand. Olhe bem para o trecho de código da **Listagem 11** que acabei de citar. Você vai reparar que, como já era de se esperar, para salvar um registro no banco de dados é necessário saber o nome de cada campo da tabela, assim como também é necessário saber o nome de cada atributo do objeto e campo da tabela onde ele será salvo.

Listagem 14. Uma parte da classe para salvar registros

```
1   unit SalvarCommand;

2   interface

3   uses Command, Bean, DB;

4   type
5    TSalvarCommand = class(TInterfacedObject, ICommand)
6    private
7     Bean: TBean;
8    public
9     constructor Create(Bean: TBean);
10    procedure Execute;
11   end;

12  implementation

13  uses DataSetFactory, Rtti;

14  { TSalvarCommand }

15  constructor TSalvarCommand.Create(Bean: TBean);
16  begin
17   inherited Create;
18   Self.Bean := Bean;
19  end;
```

Descobrir o campo onde cada atributo será salvo não é difícil, pois a fábrica cria tabelas cujos campos têm o mesmo nome dos atributos do objeto, ou seja, cada atributo será persistido em um campo cujo nome seja igual ao do atributo. Sabendo disto, fica fácil persistir as informações com o uso da nova RTTI. Tudo que a classe de comando TSalvarCommand precisa fazer para conseguir automatizar o processo de persistir objetos é descobrir o nome de cada atributo do objeto e então atribuir o valor destes atributos nos campos do TDataSet que terão o mesmo nome dos atributos, veja **Listagem 15**.

Em primeiro lugar, na linha 8 da **Listagem 15**, o método DoDataSet da fábrica é invocado para obter o objeto TDataSet já configurado. Depois disto, na linha 9, a conexão com o arquivo é estabelecida. Em seguida na linha 10 é feito um teste muito simples para descobrir se a informação que o objeto está persistindo já existe ou não. Se o registro já existir então ele deve ser aberto para edição e se ele não existir então deve ser inserido um novo registro na tabela. O teste é simples. Sabendo que o nome do campo da chave primária pode ser obtido pela constante TBean.INDEX_FIELD_NAME, então é feita uma busca no TDataSet por um registro na tabela cujo o valor da chave primária seja igual o valor do atributo ID do objeto TBean.

Listagem 15. Método Execute do comando Salvar

```
1   procedure TSalvarCommand.Execute;
2    var
3     DataSet: TDataSet;
4     RttiType: TRttiType;
5     Campos: TArray<TRttiField>;
6     Campo: TRttiField;
7   begin
8     DataSet := TDataSetFactory.DoDataSet(Bean);
9     DataSet.Open;

10    if DataSet.Locate(TBean.INDEX_FIELD_NAME, Bean.ID, []) then
11     DataSet.Edit
12    else
13     DataSet.Insert;

14    RttiType := TRttiContext.Create.GetType(Bean.ClassInfo);
15    Campos := RttiType.GetFields;
16    for Campo in Campos do
17    begin
18     if Campo.Name <> TBean.INDEX_FIELD_NAME then
19      if DataSet.FindField(Campo.Name) <> nil then
20       DataSet.FieldByName(Campo.Name).AsString
         := Campo.GetValue(Bean)
21        .ToString;
22    end;

23    DataSet.FieldByName(TBean.CLASS_NAME_FIELD).AsString :=
      Bean.ClassName;
24    DataSet.Post;
25    DataSet.Close;
26  end;
```

Após preparar o TDataSet para edição ou inserção, está na hora de alimentá-lo com as informações do objeto que está sendo persistido. É necessário obter o nome de todos os atributos para persisti-los, cada um em seu devido campo na tabela. Os recursos da RTTI são utilizados para obter o nome de cada atributo do objeto. Isto acontece na linha 14. É possível obter o nome de todos os atributos da classe. Isto acontece na linha 15: o método GetFields é invocado – ele retorna um array de objetos do tipo TRttiField.

Entre as linhas 16 e 22 é feito um loop por cada atributo da classe, e para cada atributo o campo do TDataSet cujo nome seja igual ao nome do atributo, receberá o valor do atributo em questão. Observe os testes feitos nas linhas 18 e 19. O teste da linha 18 evita que um valor seja atribuído à chave primária da tabela, uma vez que este campo é autoincremento. Já o teste da linha 19 vai garantir que o campo exista antes de tentar atribuir um valor para ele.

O processo está quase no fim. Falta dizer o nome da classe que está sendo persistida, salvar o registro e fechar a conexão com o arquivo. Estas instruções acontecem respectivamente nas linhas 23, 24 e 25.

Testando o comando Salvar

A vantagem de se ter criada uma classe de comando para automatizar o processo de salvar está no fato de que, sem automatizar o processo de salvar, o código que está entre as linhas 15 e 23 da **Listagem 11** ficaria espalhado por toda a aplicação. Se um novo campo for criado na tabela, seria um caos ter que atualizar todos os pontos do sistema onde um

registro estiver sendo salvo. Além disto tem o fato de que, um sistema comum salva diversos tipos de informações, ou seja, persiste vários objetos diferentes. Isto significa que você teria que escrever o mesmo código várias vezes um para cada tipo de objeto que for persistir. Com a classe de comando, tudo se resume em apenas uma linha, indiferente do tipo de objeto que for persistir. Além disso, se um atributo for adicionado ou excluído, você não precisa atualizar código, pois o comando de salvar vai identificar isto automaticamente.

Para testar a classe TSalvarCommand, crie uma nova unit (File > New > Unit – Delphi) no projeto TesteDaFabrica.dproj e salve esta unit com o nome de CommandInvoker.pas. Nesta unit você vai digitar o código da **Listagem 13**. Em seguida crie outra unit e salve com o nome de Command.pas. Nesta unit você vai digitar o código da **Listagem 12**. Depois crie outra unit com o nome SalvarCommand.pas e digite o código das **Listagens 14** e **15** respectivamente. Por fim, abra o arquivo DPR (View > Project Source) e escreva o código da **Listagem 16**.

Listagem 16. Testando a classe TSalvarCommand

```
1   {$R *.res}

2    var
3    Pessoa: TPessoaTest;
4    Dataset: TDataSet;

5   begin
6    Application.Initialize;

7    Pessoa := TPessoaTest.Create;
8    Pessoa.Nome := 'Paulo Miklos';
9    Pessoa.Idade := 51;

10   TCommandInvoker.Invoker(TSalvarCommand.Create(Pessoa));

11   Application.Run;
12  end.
```

Observe que a linha 10 da **Listagem 16** está substituindo todo o código que foi digitado entre as linhas 16 e 22 da **Listagem 11**. Execute o projeto e novamente você terá a impressão de que nada aconteceu, pois o sistema vai fechar quase que instantaneamente. Porém abra o arquivo PessoaTest.xml, você verá que um novo registro foi inserido no arquivo (veja **Figura 5**). Com isto pode-se garantir que a classe TSalvarCommand está funcionando sem erros.

Atualizando o agendador

Com (quase), todas as classes necessárias para persistir objetos no banco de dados prontas, falta apenas atualizar o agendador. Este processo será muito simples e rápido.

Primeiro é necessário adicionar as novas units ao projeto do agendador. As seguintes classes e interfaces que foram criadas no projeto de teste precisam ser adicionadas: TBean, TDataSetFactory, TCommandInvoker, ICommand e TSalvarCommand. Em seguida você deve alterar as classes TTarefa e TAgendamento para herdar da classe TBean. Isto significa que você deve alterar a declaração destas classes respectivamente para "TTarefa = class(TBean)" e "TAgendamento = class(TBean)", além é claro de adicionar a unit Bean.pas na sessão uses destas classes. Em seguida é necessário implementar o método GetXmlName nestas duas classes. Veja nas **Listagens 17** e **18** como ficam as implementações destes métodos nestas classes.

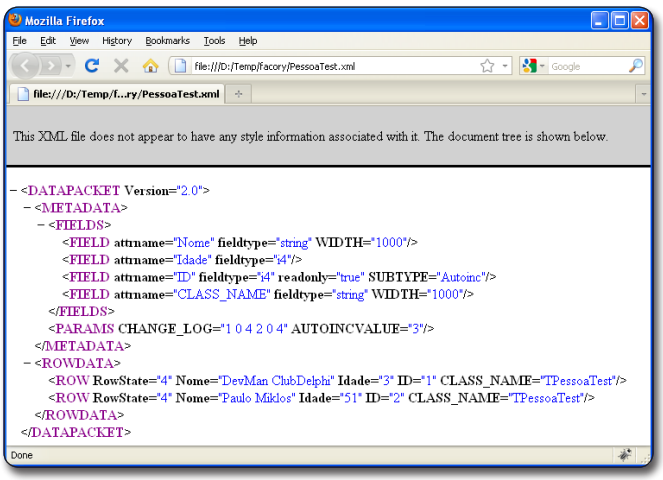


Figura 5. O novo registro inserido pelo comando de salvar

Listagem 17. Implementando GetXmlName em TAgendamento

```
1   type
2    TAgendamento = class abstract (TBean)
3    public
4     { Alguns trechos de códigos não aparecem nesta listagem }
5     function GetXmlName: string; override;
6    end;

7   implementation

8   { Alguns trechos de códigos não aparecem nesta listagem }

9   function TAgendamento.GetXmlName: string;
10  begin
11   Result := 'Agendamento.xml';
12  end;
```

Listagem 18. Implementando GetXmlName em TTarefa

```
1   type
2    TTarefa = class(TBean)
3    public
4     { Alguns trechos de códigos não aparecem nesta listagem }
5     function GetXmlName: string; override;
6    end;

7   implementation

8   { Alguns trechos de códigos não aparecem nesta listagem }

9   function TTarefa.GetXmlName: string;
10  begin
11   Result := 'Tarefa.xml';
12  end;
```

Após seguir estes passos o agendador está pronto para usar a classe de comando TSalvarCommand. Existem três momentos no agendador em que um objeto precisa ser persistido, seja ele

do tipo TTarefa ou TAgendamento. São eles: ao salvar uma tarefa, ao adicionar um agendamento para a tarefa e após executar uma tarefa. Veja na **Listagem 19** os métodos que precisam persistir os objetos.

Listagem 19. Métodos que persistem objetos

```
1   procedure TForm1.SalvarTarefaClick(Sender: TObject);
2    var
3    Tarefa: TTarefa;
4   begin
5    Tarefa := TTarefa.Create;
6    Tarefa.Comando := Comando.Text;
7    Tarefa.Nome := Nome.Text;
8    Tarefa.Parametros := Parametros.Text;
9    Tarefa.IniciarEm := IniciarEm.Text;
10   Tarefa.Comentario := Comentarios.Text;

11   Lista.Add(Tarefa);

12   MostrarTarefas;
13  end;

14  procedure TForm1.AddAgendamentoClick(Sender: TObject);
15   var
16   Ag: TAgendamento;
17  begin
18   Ag := nil;
19   case TipoAgendamento.ItemIndex of
20    AGENDAMENTO_DIARIO:
21    begin
22     Ag := TAgendamentoDiario.Create(Hora.Time);
23    end;
24    AGENDAMENTO_UNICO:
25    begin
26     Data.Time := Hora.Time;
27     Ag := TAgendamentoUnico.Create(Data.DateTime);
28    end;
29   end;
30   GetTarefaSelecinada.Agendamentos.Add(Ag);
31   MostrarTarefas;
32  end;

33  procedure TForm1.Executar(Tarefa: TTarefa; Ag: TAgendamento);
34  begin
35   ShellExecute(0, nil, PChar(Tarefa.Comando),
     PChar(Tarefa.Parametros),
36    PChar(Tarefa.IniciarEm), SW_NORMAL);
37   Ag.ÚltimaExecucao := Now;
38   MostrarTarefas;
39  end;
```

Os três métodos apresentados na **Listagem 19** estão da maneira como foram feitos na versão 2 do agendador, ou seja, ainda não estão utilizando a classe de comando TSalvarCommand para persistir os dados. Na verdade a alteração necessária é muito simples, e quase imperceptível. No método SalvarTarefaClick é necessário substituir a instrução da linha 11 por:

```
TCommandInvoker.Invoker(TSalvarCommand.Create(Tarefa));
```

Já no método AddAgendamentoClick você substitui a linha 30 por:

```
TCommandInvoker.Invoker(TSalvarCommand.Create(Ag));
```

Já no método Executar ao invés de substituir, você deve adicionar a seguinte instrução entre as linhas 37 e 38:

```
TCommandInvoker.Invoker(TSalvarCommand.Create(Ag));
```

Feito isto o agendador já estará persistindo os objetos nos devidos arquivos XML – não se esqueça de adicionar as units CommandInvoker.pas e SalvarCommand.pas na sessão uses da Form1.pas. Pode compilar, executar e testar. Quando você salvar uma nova tarefa ela não será listada no TTreeView do formulário, mas você poderá conferir diretamente no arquivo XML. No próximo tópico você vai ver como fazer para as tarefas agendadas aparecerem no TTreeView.

Listando as informações que foram persistidas

A ideia é que as informações persistidas sejam capturadas automaticamente para você. Ou seja, você não deve se preocupar com conexão e nome dos campos da tabela. Então imagine que você precise que todas as tarefas salvas sejam retornadas. Você vai precisar de duas coisas: de um objeto que saiba manipular coleções – isto já vem pronto no Delphi – e do objeto que será colecionado. Em outras palavras, você precisa de um objeto TList que vai possuir uma lista de objetos TTarefa. Cada item do objeto TList será um objeto de TTarefa que foi salvo no banco de dados. Para automatizar a busca dos objetos persistidos é necessário mais uso da RTTI, porém sem novidades, pois o processo de criar objetos a partir do banco tem uma lógica muito parecida com a de persistir objetos no banco. Veja o código da **Listagem 20**.

Listagem 20. Comando para buscar registros

```
1   unit BuscarTarefasCommand;
2   interface

3   uses Command, Tarefa, Bean, Generics.Collections, DB, Rtti;

4   type
5    TListaDeTarefas = TList<TTarefa>;

6    TBuscarTarefasCommand = class(TInterfacedObject, ICommand)
7    private
8     Resultado: TListaDeTarefas;
9     procedure CarregarAgendamentos(Tarefa: TTarefa);
10    procedure AlimentarCampos(Bean: TBean; DataSet: TDataSet;
11     Campo: TRttiField);
12   public
13    constructor Create(out Resultado: TListaDeTarefas);
14    procedure Execute;
15   end;

16  implementation

17  {TBuscarTarefasCommand}

18  uses DataSetFactory, Agendamento, SysUtils, Classes;

19  constructor TBuscarTarefasCommand.Create
    (out Resultado: TListaDeTarefas);
20  begin
21   inherited Create;
22   Resultado := TListaDeTarefas.Create;
23   Self.Resultado := Resultado;
24  end;
```

Veja na linha 13 que o construtor da classe espera por um objeto de saída do tipo TListaDeTarefas. Este objeto é na verdade um objeto TList<TTarefa>, como você pode ver na linha 5. Eu tive que criar um tipo TListDeTarefas derivado de TList<TTarefa>, pois o compilador não aceita que eu declare parâmetros out como tipo Generics. Pule diretamente para a linha 19 e veja na implementação do construtor que o parâmetro de saída Resultado é criado e em seguida a referência deste novo objeto é passada para um atributo da classe, que também é chamado de Resultado. Isto acontece, pois assim como a classe TSalvarCommand armazena uma referência ao objeto TBean que ela vai salvar, a classe TBuscarTarefasCommand armazena uma referência ao objeto TListaDeTarefas onde as tarefas encontradas serão colocadas.

O parâmetro Resultado é (re)criado na linha 22, pois por se tratar de um parâmetro out, é uma boa prática ter certeza de que o parâmetro está limpo antes de usá-lo, uma vez que você nunca sabe o que foi feito com ele antes de passá-lo como parâmetro para o construtor. Agora veja a implementação do método Execute da classe na **Listagem 21**.

Agora você vai entender como é simples buscar os registros salvos, criar e alimentar os atributos dos objetos das classes que cada registro representa. A lógica é simples. Se cada campo da tabela tem o mesmo nome do atributo da classe, então o que deve ser feito é pegar o nome de todos os atributos da classe e iterar com cada um destes atributos, buscando o valor de um campo na tabela que tenha o mesmo nome do atributo.

Como pode ver na **Listagem 21**, mais códigos da RTTI são utilizados, porém não há muita diferença do que você já aprendeu. Lembre-se que o nome da classe de comando é TBuscarTarefasCommand, portanto é óbvio que ela vai buscar por registros da classe TTarefa. Em primeiro lugar, na linha 9 é criado um objeto da classe TTarefa. Depois o método DoDataSet da fábrica é invocado para criar o TDataSet para estabelecer conexão com o arquivo onde as informações persistidas dos objetos TTarefa estão, veja na linha 10. Em seguida, após abrir o TDataSet retornado pela fábrica, eu peço para a classe TRttiContext retornar a estrutura RTTI da classe TTarefa. Como já disse, você consegue isto invocando o método GetType da classe TRttiContext. Com a estrutura RTTI em mão, basta pegar a lista de atributos desta estrutura. Isto acontece na linha 13. Um loop inicia-se na linha 14. Este loop só vai parar quando chegar ao último registro da tabela de dados. Para cada registro que o loop encontrar um objeto TTarefa é criado e povoado com as informações do registro em questão. Veja, na linha 16 que a tarefa é criada e na linha 17 é criado um segundo loop para cada atributo da classe TTarefa. Para cada atributo encontrado é feita uma chamada ao método AlimentarCampos. A implementação deste método está na linha 25, mas preste atenção que ele recebe três objetos como parâmetros, são eles: Tarefa, DataSet e Campo (veja linha 18).

Este método é realmente simples. Veja entre as linhas 28 e 40 que ele vai buscar no registro atual do objeto DataSet o valor do campo que tenha o mesmo nome do atributo Campo (variável do loop na linha 17) e alimentar o atributo correspondente do objeto Tarefa com o valor do campo no TDataset. Em outras palavras ele vai pegar, por exemplo, o valor do campo IniciarEm que está no arquivo XML e colocar dentro do atributo também chamado de IniciarEm da classe TTarefa. É claro que para fazer isto o método AlimentarCampos leva em consideração o tipo de informação que o campo armazena. Por isto existe a instrução "case" da linha 30.

Quando o loop da linha 17 terminar significa que todos os atributos da classe TTarefa já foram alimentados com as informações do registro para o qual o DataSet está apontando. Feito isto agora está faltando carregar os agendamentos desta

Listagem 21. Buscando os registros

```
1  procedure TBuscarTarefasCommand.Execute;
2   var
3    DataSet: TDataSet;
4    Tarefa: TTarefa;
5    RttiType: TRttiType;
6    Campos: TArray<TRttiField>;
7    Campo: TRttiField;
8  begin
9    Tarefa := TTarefa.Create;
10   DataSet := TDataSetFactory.DoDataSet(Tarefa);
11   DataSet.Open;

12   RttiType := TRttiContext.Create.GetType(Tarefa.ClassType);
13   Campos := RttiType.GetFields;
14   while not DataSet.Eof do
15   begin
16    Tarefa := TTarefa.Create;
17    for Campo in Campos do
18     AlimentarCampos(Tarefa, DataSet, Campo);

19    CarregarAgendamentos(Tarefa);
20    Resultado.Add(Tarefa);
21    DataSet.Next;
22   end;
23   DataSet.Close;
24  end;

25  procedure TBuscarTarefasCommand.AlimentarCampos
    (Bean: TBean; DataSet: TDataSet;
26   Campo: TRttiField);
27  begin
28   if DataSet.FindField(Campo.Name) <> nil then
29   begin
30    case DataSet.FieldByName(Campo.Name).DataType of
31     ftString:
32      Campo.SetValue(Bean, DataSet.FieldByName
        (Campo.Name).AsString);
33     ftAutoInc, ftInteger:
34      Campo.SetValue(Bean, DataSet.FieldByName
       (Campo.Name).AsInteger);
35     ftFloat:
36      Campo.SetValue(Bean, DataSet.FieldByName
       (Campo.Name).AsFloat);
37    else
38     raise ENotSupportedException.Create('Tipo não suportado');
39    end;
40   end;
41  end;
```

tarefa. Isto é feito na linha 19 com uma chamada ao método CarregarAgendamentos. Este método recebe como parâmetro o objeto TTarefa cujos agendamentos devem ser carregados. Após carregar os agendamentos, o objeto TTarefa criado na linha 16 está pronto para ser usado, pois todas as suas informações já foram carregadas do banco de dados. Assim sendo, a tarefa é adicionada na lista de resultados (linha 20) e o ponteiro do DataSet passa para o próximo registro (linha 21).

Esta lista de resultados usada na linha 20 nada mais é do que o objeto TListaDeTarefas que foi passado no construtor da classe – você viu isto na **Listagem 20**. Você deve lembrar que quando o comando acabar sua execução todas as tarefas encontradas estarão dentro desta lista. Ainda falta um método: veja a implementação do método CarregarAgendamentos na **Listagem 22**.

Nota

Observe o tipo da variável declarada na linha 5 da **Listagem 22**. O tipo TAgendamentoClass foi declarado na unit Agendamento.pas, logo abaixo da palavra reservada type da unit. Baixe o exemplo disponível para download e confira. Você verá que o construtor da classe TAgendamento também foi modificado: agora para criar um agendamento é necessário especificar o objeto TTarefa ao qual o agendamento pertence. Você vê isto acontecendo na linha 10 da **Listagem 22**.

Listagem 22. Carregando os agendamentos de uma tarefa

```
1   procedure TBuscarTarefasCommand.CarregarAgendamentos
    (Tarefa: TTarefa);
2    var
3    DataSet: TDataSet;
4    Agendamento: TAgendamento;
5    ClasseDeAgendamento: TAgendamentoClass;
6    RttiType: TRttiType;
7    Campos: TArray<TRttiField>;
8    Campo: TRttiField;
9   begin
10   Agendamento := TAgendamento.Create(Tarefa);
11   DataSet := TDataSetFactory.DoDataSet(Agendamento);
12   DataSet.Filter := 'IdTarefa = ' + IntToStr(Tarefa.ID);
13   DataSet.Filtered := True;
14   DataSet.Open;

15   RttiType := TRttiContext.Create.GetType(Agendamento.ClassType);
16   Campos := RttiType.GetFields;
17   while not DataSet.Eof do
18   begin
19    ClasseDeAgendamento := TAgendamentoClass
20     (FindClass(DataSet.FieldByName
       (TBean.CLASS_NAME_FIELD).AsString));
21    Agendamento := ClasseDeAgendamento.Create(Tarefa);
22    for Campo in Campos do
23     AlimentarCampos(Agendamento, DataSet, Campo);

24    Tarefa.Agendamentos.Add(Agendamento);
25    DataSet.Next;
26   end;
27   DataSet.Close;
28  end;
```

Como já foi dito, o método CarregarAgendamentos tem a responsabilidade de encontrar todos os agendamentos de uma tarefa e então carregá-los para o atributo Agendamentos da classe TTarefa. É exatamente isto que ele faz. Como já era de se esperar ele é muito parecido com o método Execute. A única diferença é que o método execute vai carregar informações da classe TTarefa enquanto que este método vai carregar as informações da classe TAgendamento.

Por serem muito parecidos, não há necessidade de re-explicar o código. Vou apenas destacar o que está acontecendo nas linhas 12 e 13, pois nestas linhas é feito o filtro para garantir que somente os agendamentos referentes à tarefa recebida como parâmetro do método serão retornados quando o DataSet for aberto na linha 14.

Pronto. A classe de comando TBuscarTarefasCommand terminou. Agora falta modificar apenas mais uma linha da classe TForm1 do agendador para que utilize a classe TBuscarTarefasCommand e tudo estará concluído.

A última alteração

Observando o código da classe TForm1 do agendador mais detalhadamente, você vai perceber que toda a classe está funcionando com base na variável Lista – aquela variável que era usada até agora como se fosse o banco de dados do agendador. Se eu remover a variável Lista da classe, terei que mudar muita coisa dentro dela, pois toda a classe TForm1 se baseia na existência desta variável.

Então ao invés de removê-la eu vou apenas mudar a maneira como ela é manipulada. Depois da alteração feita no tópico Atualizando o agendador, a variável já não está mais sendo alimentada. Isto significa que você pode agendar quantas tarefas quiser, mas elas nunca serão executadas, pois o TTimer que vai executar a tarefa na hora correta está olhando para as tarefas que estão na variável Lista. Porém, agora as tarefas estão sendo adicionadas no arquivo XML ao invés da variável Lista, nunca terá uma tarefa agendada dentro da variável Lista e por isto o TTimer nunca vai executar nada.

Para corrigir isto e fazer com que as tarefas salvas no arquivo XML sejam carregadas para a variável Lista eu vou mudar o método MostrarTarefas da classe TForm1. Este método tem a função de iterar com todas as tarefas dentro da variável Lista e exibi-las dentro do TTreeView do Form1. O problema é que a variável Lista nunca terá tarefas para serem exibidas, então antes de começar a iteração com a variável eu vou fazer uma chamada ao comando buscar para que todas as tarefas sejam carregadas. Isto vai resolver o problema e sempre que eu quiser atualizar o conteúdo da variável Lista, para que ela tenha todas as tarefas salvas no arquivo XML, basta fazer uma chamada ao método MostrarTarefas.Veja como era o código do método MostrarTarefas antes da utilização da classe de comando TBuscarTarefasCommand, na **Listagem 23**.

A **Listagem 23** mostra como o método MostrarTarefas era até agora, ou seja, ele não está atualizando a variável Lista com as informações do arquivo XML. Para conseguir isto basta adicionar a instrução "TCommandInvoker. Invoker(TBuscarTarefasCommand.Create(Lista));" entre as linhas 9 e 10. Isto vai fazer com que a variável Lista seja atualizada antes do loop começar. Depois você deve criar a classe

Listagem 23. O antigo método MostrarTarefas

```
1   procedure TForm1.MostrarTarefas;
2    var
3    Tr: TTarefa;
4    Ag: TAgendamento;
5    ProximaExecucao, ÚltimaExeucao, Aux: TDateTime;
6    LinhaSelecionada: Integer;
7   begin
8    LinhaSelecionada := ListaDeTarefas.ItemIndex;
9    ListaDeTarefas.Clear;
10  for Tr in Lista do
11  begin
12   with ListaDeTarefas.Items.Add do
13   begin
14    Caption := Tr.Nome;
15    SubItems.Add(Tr.Comando + '' + Tr.Parametros);

16    if Index = LinhaSelecionada then
17    begin
18     Selected := True;
19     ListaDeTarefasClick(Self);
20    end;
21    ProximaExecucao := 0;
22    ÚltimaExeucao := 0;
23    for Ag in Tr.Agendamentos do
24    begin
25     Aux := Ag.GetDataDaProximaExecucao;
26     if (ProximaExecucao = 0) or ((Aux > 0) and
      (Aux < ProximaExecucao)) then
27      ProximaExecucao := Aux;

28     if Ag.ÚltimaExecucao > ÚltimaExeucao then
29      ÚltimaExeucao := Ag.ÚltimaExecucao;
30    end;
31    SubItems.Add(IfThen(ProximaExecucao = 0,'<nunca>',
      DateTimeToStr
32       (ProximaExecucao)));
33    SubItems.Add(IfThen(ÚltimaExeucao = 0,'<nunca>',
      DateTimeToStr
34       (ÚltimaExeucao)));
35   end;
36  end;
37 end;
```

TBuscarTarefasCommand. Para isto, crie uma nova unit no Delphi 2010 e salve-a com o nome BuscarTarefasCommand.pas. Depois digite nesta unit o código das **Listagens 20, 21** e **22** respectivamente. Por último adicione a unit BuscarTarefasCommand.pas na sessão uses da Form1.pas. Pronto! Tudo está funcionando corretamente. As tarefas estão sendo agendadas e executadas no horário programado. Só não direi que você pode compilar e executar, pois como já foi dito na última nota, existe uma diferença no construtor da classe TAgendamento. Baixe o exemplo e confira a diferença.

Conclusão

Agora o agendador chegou a sua última versão. Ainda existem alguns problemas na classe TForm1, afinal de contas ela ainda é responsável por fazer o monitoramento dos agendamentos para executá-los na hora correta. Esta responsabilidade precisa ser tirada da classe TForm1 e posta em uma nova classe exclusiva para isto. Neste caso a classe que vai fazer este monitoramento não será uma classe de comando, ao invés disto você pode criar uma classe que encapsule um TTimer e através do padrão Observer fazer com que as classes de agendamento sejam notificadas quando chegar a hora agendada. Além disto, existem outros problemas evidentes, como por exemplo, problemas de desempenho, pois o método MostrarTarefas é invocado excessivamente em diversos pontos da classe TForm1, além do fato de que talvez seja necessário criar um algoritmo de pesquisa para que o usuário possa encontrar uma tarefa específica – você pode conseguir isto com a implementação do padrão Strategy em conjunto com o padrão Command – que já foi implementado no projeto.

Porém esqueça os problemas e concentre-se no que esta série de artigos tentou passar. Você aprendeu como é simples usar a nova RTTI do Delphi, entendeu como é o funcionamento de um framework (você criou um mini-framework) e teve a oportunidade de ver como um verdadeiro sistema OO deve ser arquitetado. Baixe a última versão do agendador disponível para download e tente resolver os problemas apresentados no parágrafo anterior. Boa diversão! ●

Links

Artigo Clube Delphi 111 - Delphi 2010 – A nova RTTI
http://www.devmedia.com.br/articles/viewcomp.asp?comp=14774

Artigo Clube Delphi 113 - Padrão Command
http://www.devmedia.com.br/articles/viewcomp.asp?comp=15246

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Delphi Prism e ASP.NET

Reaproveite seus conhecimentos Win32 / Desktop para criar aplicações Web – Parte 1

Guinther Pauli

guinther@devmedia.com.br
guinther.pauli@gmail.com
Blogs: http://guintherpauli.blogspot.com
http://twitter.com/guintherpauli
http://gpauli.com

Tem 22 anos de experiência na área de desenvolvimento e arquitetura de software. Certificado em Delphi 3, 5, 6, 7, 2005, 2006, Delphi para Web e Delphi para Linux, certificado Microsoft MCP, MCAD, MCSD.NET, MCTS e MCPD (C#, ASP.NET). Editor da Revista ClubeDelphi e da Revista .NET Magazine Brasil, professor de Pós-Graduação da UNIVEM-SP. Especialista em desenvolvimento Delphi, C#, .NET, ASP.NET, atuando também em projetos de migração orientando empresas e desenvolvedores na adoção de melhores práticas. Foi arquiteto e desenvolvedor do site de doações on-line do projeto Criança Esperança da Rede Globo / Unesco.

Do que trata o artigo

Lançado pela Embarcadero, o Delphi Prism é a nova forma de desenvolver aplicações para o .NET Framework e Mono utilizando o Object Pascal e o IDE do Visual Studio. Nesta série de artigos, veremos como utilizar o Delphi Prism para criar um site de E-Commerce usando a tecnologia ASP.NET.

Para que serve

O objetivo principal desta série de artigos é familiarizar os desenvolvedores Delphi Win32 com o ambiente de desenvolvimento do Delphi Prism, o IDE do Visual Studio, bem como introduzir o desenvolvimento de aplicações Web. Dessa forma, faremos vários comparativos entre os dois IDE's e também entre os Frameworks ASP.NET e VCL, servindo como um excelente guia para diminuir a curva de aprendizado ao se aprender ASP.NET baseando-se nos seus prévios conhecimentos em Delphi Win32 (Desktop).

Em que situação o tema é útil

Esta série será muito útil para quem nunca desenvolveu aplicativo algum para Web, nem mesmo teve um primeiro contato com o Delphi Prism / Visual Studio. Ao desenvolver aplicações para Web, ganhamos inúmeras vantagens se compararmos ao ambiente Desktop: deploy simplificado, uso de computação nas nuvens, não há necessidade de instalações visto que os usuários acessam um mesmo servidor na internet, não há dependência do sistema operacional pois usamos um browser, não há preocupação com backups nos clientes, nem altos custos para manutenções e visitas aos clientes etc.

Resumo do DevMan

Este artigo é o primeiro de uma série que vai mostrar como desenvolver um Web Site utilizando as tecnologias ASP.NET e Dephi Prism. Nesta primeira etapa, teremos uma introdução ao Delphi Prism, seus objetivos e propósitos, o que também servirá de motivação para aqueles que vão iniciar os trabalhos no desenvolvimento para Web. A seguir, partiremos para a criação do banco de dados no SQL Server. Mais do que desenvolver um Web Site, o principal objetivo desta série sem dúvida será orientá-lo, com dicas preciosas e precisas, de como reaproveitar seus conhecimentos em Delphi Win32, RAD Studio, desenvolvimento Desktop, VCL para criar aplicações em Delphi Prism, Visual Studio, Web e .NET Framework. Não se preocupe,

O .NET é a plataforma mais consolidada e robusta para o desenvolvimento de aplicações Web. Ela faz parte do dia-a-dia do desenvolvedor Delphi desde o Delphi 8, quando foi apresentada a tecnologia no então novo IDE da Borland. O Delphi for .NET, como ficou conhecida na época a versão do compilador que suportava a nova plataforma da Microsoft, foi descontinuada em 2009, quando fomos apresentados a um novo produto, muito mais produtivo, estável e consolidado.

O Delphi Prism já nasceu grande por dois motivos: ele utiliza um IDE não adaptado para desenvolvimento para o .NET, como foi o BDS (Borland Developer Studio), mas o IDE do Visual Studio, especializado para este propósito, que tem uma grande equipe que cuida somente do ambiente. Ou seja, isso livra a equipe do Delphi da evolução do IDE, que pode então cuidar da evolução do seu outro IDE, o RAD Studio, para Win32, PHP (e em breve Mac e Linux). Tão logo saia uma versão nova do VS, em poucos dias já teremos uma versão do Delphi Prism para ele, podendo desfrutar de todos os novos recursos por ele proporcionados e as novas tecnologias que surgem no .NET Framework. Nós vamos comprovar isso na prática, quando for lançado o Visual Studio 2010. É bem provável que quando você estiver lendo este artigo, o VS2010 versão final já esteja no mercado, pois seu lançamento está previsto para 12 de Abril. Em maio já teremos uma versão do Prism para o VS2010.

E segundo, ele é um produto que já existe há muito tempo, então já está maduro e estável. Ele foi desenvolvido pela RemObjects, uma empresa parceira da Borland, que criou uma versão do Object Pascal para o .NET e usava o IDE do VS, sendo assim, concorrente direto do Delphi for .NET. Foi então que a Embarcadero adquiriu os direitos de uso de propriedade do produto então chamado Chrome, e rebatizou-o com o intuitivo e comercial nome de Delphi Prism. Todos saímos ganhando, e muito.

Sim, há uma curva de aprendizado ao você usar um outro IDE, se já está bem acostumado ao Delphi Win32 (vou usar este termo para me referenciar daqui para frente ao Delphi "tradicional"). Porém, muitas coisas são praticamente idênticas. Por exemplo, no VS o Project Manager se chama Solution Manager. O Object Inspector se chama Property Window, o Data Explorer tem um equivalente, chamado Server Explorer. A paleta de componentes no VS se chama Tool Box. O editor de código é idêntico, possui inclusive um Code Insight, que aqui se chama IntelliSense. O designer, principalmente para o ASP.NET, é um pouco mais pesado que o da VCL Forms (você verá isso na prática).

O uso de ASP.NET Server Controls proporciona um desenvolvimento intuitivo, simples, muito similar ao drag & drop que estamos acostumados na VCL. Temos os mesmos conceitos e aplicabilidades, cada controle tem suas propriedades, métodos, eventos, se ligam a outros componentes, como na VCL. Muitos componentes têm os mesmos nomes. O acesso a dados muda um pouco, claro, muito devido ao paradigma Web. O ASP.NET foi concebido, já na sua primeira versão, com um propósito bem definido: facilitar a vida do desenvolvedor Desktop. Isso porque o desenvolvimento para Web, pré ASP.NET, era algo realmente arcaico, o que desencorajou muitos a optar por uma tecnologia. ASP e PHP clássico, por exemplo, mesclavam código de apresentação (layout), acesso a dados e regras de negócio em um único arquivo. Orientação a objetos zero. Boas práticas inexistentes. Isso não existe no ASP.NET.

E finalmente, a linguagem do Delphi Prism é muito semelhante ao Delphi tradicional, eu diria melhor, pois ela quebra algumas normas rígidas do Pascal, porém sem perder sua elegância, ao mesmo tempo que padroniza alguns termos e conceitos já bastante conhecidos e utilizados por outras linguagens orientadas a objetos, como C# e Java. Exemplos são o suporte a type inference, namespaces, generics, object initializers, o operador new, foreach etc. Eu sei, muitos desses recursos começaram a ser implementados no Delphi Win32, e eu espero que um dia possamos contar com todos eles.

Também há uma curva de aprendizado ao se partir do ambiente Desktop para Web, e acho que aqui é que mora o problema. Muitas coisas que não nos preocupam no Windows são extremamente importantes no ASP.NET. Por exemplo, cada byte que você aloca no servidor e por descuido deixa-o "esquecido", vai ter um impacto enorme ao final do dia. Performance é fator crucial e pré-requisito indispensável no seu projeto. Acredite, você vai usar muito o try finally daqui para frente. Ou melhor, o using, suportado pelo Delphi Prism e que possui equivalência ao using do C# (não confunda com o using dos namespaces, esse equivale ao uses). Postbacks, gerenciamento de estado, a não presença de janelas modais, a necessidade de uso de JavaScript, CSS, AJAX, tudo isso vai impactar na produtividade da equipe Delphi ao criar as primeiras aplicações para Web.

Dessa forma, essa série não será apenas um mero tutorial que vai levar você do início ao fim mostrando como construir uma aplicação completa para Web. Nesta série, nós vamos de fato minimizar ao máximo a sua curva de aprendizado ao adotar o Prism para desenvolvimento Web, de forma que vou fazer comparativos durante todo o desenvolvimento do exemplo. Dessa forma, essa série vai servir para você:

- Aprender .NET Framework e ASP.NET reaproveitando seus conhecimentos na VCL;
- Aprender Delphi Prism (que eu prefiro chamar carinhosamente de D#) reaproveitando seus conhecimentos em Object Pascal;
- Aprender a usar o Visual Studio como IDE de forma semelhante ao que você já conhece utilizando o IDE do Delphi Win32;
- Criar aplicações para Web com focos nos cuidados para quem parte do ambiente Desktop, ou seja, focaremos nas diferenças das quebras de paradigmas;
- Usar o SQL Server como banco de dados reaproveitando conhecimentos adquiridos ao utilizar o Firebird. É possível usar o Firebird com ASP.NET, porém, já advirto que o SQL Server é o banco mais rápido e robusto para uso com ASP.NET, já que são da mesma "família";

Cada parte deste curso vai cuidar do desenvolvimento de uma parte da aplicação, do nosso Web Site. Vamos começar pela criação do banco de dados, depois vamos criar a estrutura básica do site e navegação, faremos as páginas de pesquisas, cadastros, carrinho de compras, check-out, implementaremos a parte de segurança completa e finalmente o deploy.

Este artigo não utilizará nenhuma metodologia específica nem padrão específico, apesar de existirem muitas alternativas no mercado, principalmente quando se fala em .NET. Eu vou deixar o nosso exemplo o mais parecido com o que você já faz no Delphi Win32, justamente para incentivá-lo a adotar o ASP.NET como plataforma de desenvolvimento Web para seus próximos projetos, e nem imaginar em usar outras tecnologias inferiores (e você vai comprovar isso). Dessa forma, não vamos criar um exemplo usando camadas (multicamadas), nem padrões como MVC (já existe o ASP.NET MVC), tão pouco usar um framework de mapeamento objeto / relacional (hoje os mais populares são o NHibernate e o ADO.NET Entity Framework). Eu abordarei estes temas em outros artigos aqui na ClubeDelphi, já quando o Visual Studio 2010 estiver em produção. Vamos à prática.

Nota

No .NET Framework, como veremos em futuros artigos, ao utilizar um framework de mapeamento O/R como o ADO.NET Entity Framework, temos a opção de começar o desenvolvimento da aplicação pela modelagem do domínio, pelo diagrama de classes, e não pelo modelo relacional do banco de dados. É o que chamamos de Model First Development. Essa também é uma das premissas do DDD – Domain-Driven Design, que diz que devemos focar primeira e exclusivamente nos problemas que o sistema se propõe a resolver, e não nos problemas de infraestrutura e tecnologia. Neste artigo, como nosso intuito não é acentuar a curva de aprendizado para quem vem do Win32, vamos começar da forma tradicional, pela modelagem E/R.

Preparando o ambiente

Para construir esta série, você vai precisar do Delphi Prism, usaremos aqui a versão que suporta o Visual Studio 2008. Se você já possui o VS2008 instalado, o Delphi Prism irá instalar apenas seus plug-ins e o suporte para a linguagem Oxygene (o compilador do Prism). Se não tem, quando instalar o Delphi Prism ele também instalará o VS2008. Lembrando que o Prism não roda na versão gratuita do VS, o Web Developer Express, por uma limitação da Microsoft que não permite que outras linguagens sejam adicionados ao IDE nas versões Express de seus produtos. Então, o Prism só funciona com VS Professional ou superior.

Como banco de dados utilizaremos a versão gratuita do SQL Server 2005 (a versão que vem com o VS2008), chamada de SQL Server Express. Quando instalamos o Delphi Prism / Visual Studio, por padrão o SQL Server Express já é instalado de fundo e já sai rodando. Porém, ele não vem com um gerenciador (front-end) para você administrá-lo, apesar da maioria das tarefas relativas à manutenção de um banco de dados poderem ser feitas de dentro do próprio IDE. Dessa forma, vamos precisar baixar e instalar o Microsoft SQL Server Management Studio Express.

Na primeira etapa desta série não será necessário possuir um servidor Web instalado na máquina, como o IIS (Internet Information Services). Desde o Visual Studio 2005, o ASP.NET faz uso de um servidor local que é usado na etapa de desenvolvimento, que roda como um processo e faz o papel do Web Server na porta 80. Para quem já usou o Delphi for .NET antes, ele equivale ao Cassini. Aliás, isso facilitou a vida de todos, visto que para criar um site e rodá-lo, basta apertar F5 (nada de deploy). Para abrir um existente, basta dar um Open no Web Site e rodar. Como aplicações Delphi tradicionais. Antes (VS2003), era preciso ter conhecimentos de IIS, criar diretórios virtuais etc.

Você encontra na sessão Links os endereços para downloads de todos estes itens.

Criando o banco de dados

Abra ao Microsoft SQL Server Management Studio Express. Na janela de diálogo que aparece (**Figura 1**) informe em Server o nome do servidor SQL Server e a sua instância, por padrão .\SQLEXPRESS, onde o ".\" significa que estamos acessando um servidor local (equivale ao famoso "localhost" do Firebird) e SQLEXPRESS é o nome da instância. Caso for acessar um servidor em outro local, utilize o formato NOME_DO_SERVIDOR_OU_IP\INSTANCIA. Em Authentication, mantenha o padrão Windows Authentication, com isso usaremos o mesmo usuário que está logado no Windows atualmente para acessar o servidor SQL Server, o que também chamamos de autenticação integrada. Clique em Connect.

Figura 1. Conectando ao SQL Server Express

Será aberto o ambiente principal do Microsoft SQL Server Management Studio Express. À esquerda temos o Object Explorer, onde podemos navegar pelos objetos de um banco de dados, como tabelas, procedures, views etc. Neste painel, clique de direita sobre o item Databases, escolhendo então New Database (**Figura 2**).

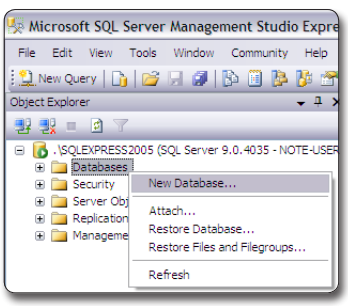


Figura 2. Opção Databases > New Database para criar um novo banco de dados

Nota

Em um ambiente real de produção, você normalmente não colocará o banco de dados no mesmo local onde está o seu Web Site, isso porque o servidor Web é uma máquina que tem um endereço acessível para qualquer usuário na Web. Por questões de segurança, o banco deve ficar em um local dentro da intranet acessível somente ao servidor Web e estações locais, mas não exposto a toda a internet.

Na janela que aparece, chamada New Database, vamos informar o valor LOJA como sendo o nome do nosso novo banco de dados, na opção Database name (**Figura** 3). Observe que o SQL Server permite que você informe o caminho dos arquivos a serem criados para este novo banco de dados, o seu tamanho inicial etc. Normalmente não trabalhamos diretamente com o arquivo físico gerado pelo SQL Server, mas sim com alias que informamos.

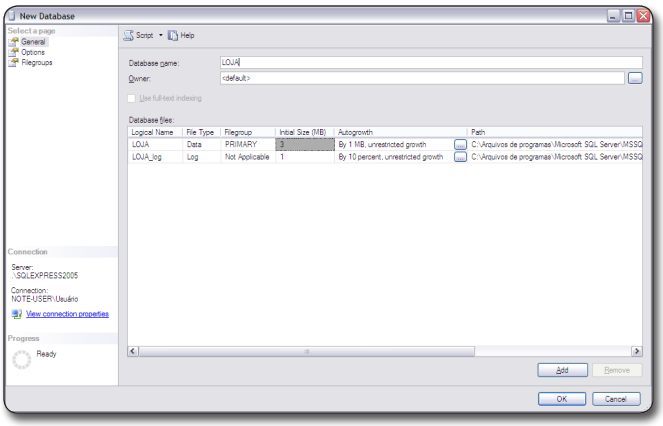


Figura 3. Criando um banco de dados

Criando as tabelas do banco de dados

Agora no Object Explorer aparecerá o nosso Database LOJA, expanda-o, clique de direita sobre o item Tables e escolha New Table. Será aberto um designer onde você poderá criar uma nova tabela. Vamos começar pela tabela mais simples do sistema, a Categorias de produtos. Preencha os campos da estrutura conforme mostra a **Figura 4**.

Observe em especial o campo CategoryID. Ele é do tipo inteiro, auto-incremento e chave-primária. Para configurá-lo dessa forma, selecione o campo e clique no ícone da chave na barra de ferramentas, ou ainda clique de direita sobre o campo e escolha Set Primary Key. Para que seus valores sejam incrementados automaticamente, basta ir até a janela de propriedades e definir a opção Indentity Specification > (Is Identity) como True. Também é possível especificar o valor inicial da numeração e o valor de cada incremento, vamos manter o padrão que é o valor 1 para ambos. Note que isso equivale a usar um Generator / Sequence no Firebird. Ambos os campos criados não podem receber valores nulos, logo desmarcamos o campo Allow Nulls. Clique no ícone do disquete na barra de ferramentas para salvar a tabela, com o nome de Categories.

De forma semelhante, proceda a criação das demais tabelas do banco de dados, chamadas Products, Orders e OrderDetails. Para facilitar a visualização da estrutura dos campos, dados e relacionamentos, veja o diagrama da **Figura 5**. Você pode usar o próprio designer de criação de diagramas para criar as demais tabelas, para isso, basta clicar de direita no Object Explorer sobre o item Database Diagrams, confirmar a mensagem e escolher New Database Diagram. A seguir, clique de direita na área branca e escolha New Table.

Para criar os relacionamentos entre as tabelas, basta selecionar o campo chave estrangeira de uma tabela e arrastar para

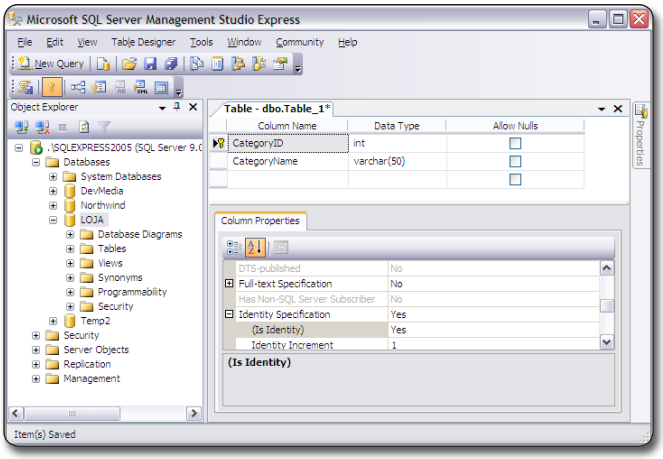


Figura 4. Criando uma tabela no SQL Server

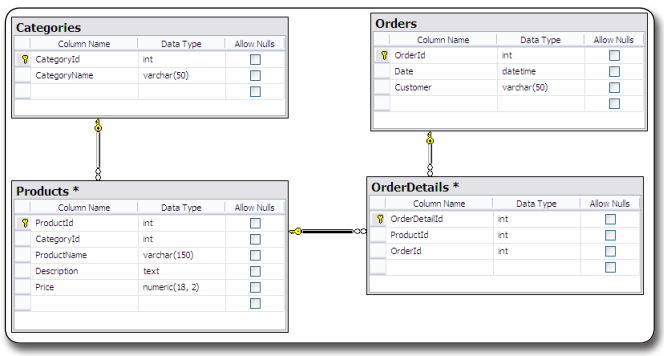


Figura 5. Diagrama do banco de dados

o campo chave primária da tabela relacionada. Por exemplo, o campo ProductID da tabela OrderDetails (detalhes do pedido) precisa se relacionar com o respectivo campo ProductID da tabela de Produtos. Para isso basta arrastar o campo da tabela OrderDetails para Products para que apareça a janela mostrada na **Figura 6**, onde confirmamos o nome do relacionamento e os campos relacionados. Também é possível configurar as opções de integridade, por exemplo, para as ações Update Rule e Delete Rule. Nesse caso vamos manter "No Action", pois quando excluirmos um produto, não queremos que todos os itens de pedidos já feitos sejam excluídos (o que seria um Cascade), dessa forma mantemos nossos dados íntegros.

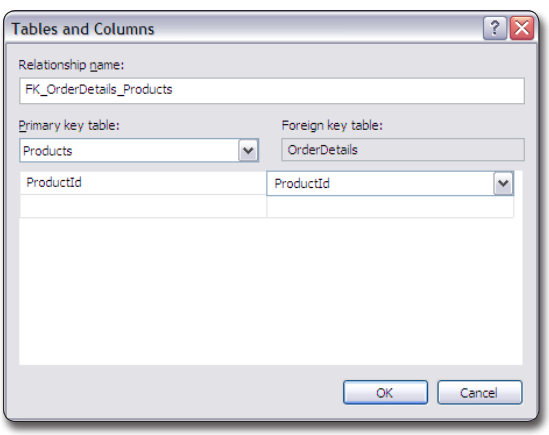


Figura 6. Configurando o relacionamento entre Foreign Key e Primary Key

Nota

Repare que a estrutura do nosso banco de dados é extremamente simples e muitos campos e tabelas que estão presentes em sistemas de E-Commerce reais não foram modelados aqui. Apesar de construirmos um Web Site do início ao fim, nosso foco aqui não é estender o sistema, mas conduzi-lo com harmonia a criar suas primeiras aplicações Web, com objetivo de aprender as tecnologias ASP.NET, Delphi Prism e Visual Studio.

Inserindo dados para teste

Vamos popular nosso banco de dados com algumas informações para facilitar os testes quando formos criar o Web Site no Delphi Prism. Clique de direita sobre o nome da tabela Categories e escolha a opção Open Table. Adicione algumas categorias conforme mostra a **Figura 7.** Repita o mesmo procedimento para a tabela de Produtos, inserindo alguns registros conforme mostra a **Figura 8**. Lembre-se de respeitar o valor para a Foreign Key CategoryID, referenciando o ID da categoria correspondente.

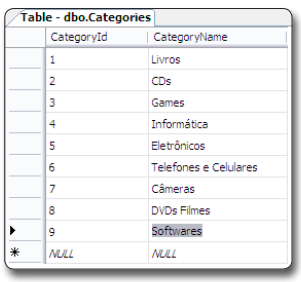


Figura 7. Inserindo dados na tabela de Categorias

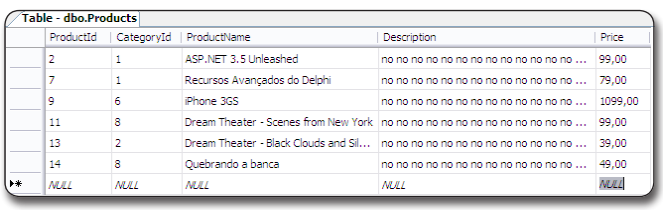


Figura 8. Inserindo dados na tabela de Produtos

Configurando a conexão no Delphi Prism / Visual Studio

Criado nosso banco de dados, vamos configurar a conexão para o mesmo no IDE do Delphi Prism, o Visual Studio 2008. Com o IDE aberto, clique no menu View > Server Explorer. Esta janela permite que o desenvolvedor navegue pelos vários recursos disponíveis no servidor local, ou mesmo remotos, podendo administrar a partir do próprio IDE serviços, logs de eventos, bancos de dados etc.

Clique de direita no item Data Connections e escolha Add Connection (**Figura 9**). Na janela que abre (**Figura 10**) coloque os parâmetros de conexão ao banco, informando o nome do servidor e instância em Server name (.\SQLEXPRESS) e escolhendo o banco de dados LOJA em "select or enter a database name".

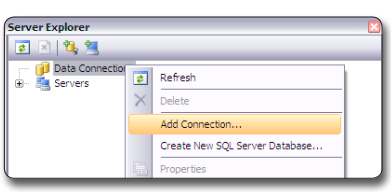


Figura 9. Adicionando uma conexão no IDE

Feito isso, uma nova conexão estará configurada e disponível no IDE. Note que isso não configura uma conexão para uma aplicação em particular, é uma conexão persistente que ficará disponível no IDE para qualquer tipo de aplicação (algo como o Data Explorer do Delphi Win32). Podemos inclusive usar o próprio IDE do Delphi Prism / Visual Studio para navegar pelos objetos do banco (**Figura 11**) e realizar praticamente todas as operações de administração que são feitas no Microsoft SQL Server Management Studio Express, como criar tabelas, diagramas, executar comandos SQL, criar procedures, visualizar dados e até mesmo depurar a execução de Stored Procedures.

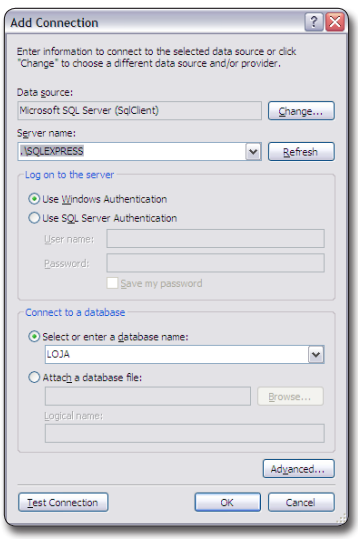


Figura 10. Parâmetros de conexão

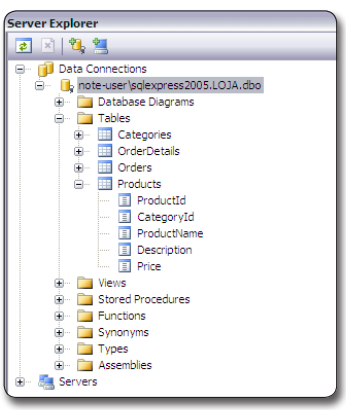


Figura 11. É possível administrar o banco de dados de dentro do próprio IDE

Testando

Neste ponto, antes de finalizarmos a primeira parte deste artigo e ficarmos prontos para o restante da série, é ideal fazer um teste para nos certificarmos que tudo está rodando e configurado corretamente. Não iremos iniciar a construção do site ainda, mas apenas criar uma página para exibir os dados da tabela de produtos no browser, o que também ajudará você a já ir se familiarizando com o desenvolvimento Web com ASP.NET no Delphi Prism / Visual Studio.

Nota do DevMan

A opção Data Source, mostrada na parte superior desta janela, é muito importante, pois corresponde ao provider do ADO.NET que será usado para a conexão (falaremos mais sobre ADO.NET adiante neste curso). Aqui estamos usando o provider nativo para o SQL Server (mais rápido e otimizado), chamado Microsoft SQL Server (SqlClient). Note que podemos aqui usar outros providers para acessar outros bancos de dados, como Interbase, Blackfish SQL, Access, bancos com drivers ODBC ou OLEDB (jamais os use em aplicações Web) e até mesmo servidores DataSnap.

Clique em File > New Web Site (o que vai abrir uma janela semelhante ao Object Repository do Delphi Win32), escolha o template ASP.NET Web Site, digite a localização do site em Location (ex.: "C:\Prism"), defina a linguagem a ser utilizada em Language (escolha Oxygene que é a linguagem / compilador do Delphi Prism) e clique em OK.

No designer aberto, clique no botão Design logo na parte inferior. A partir da Tool Box (equivalente à Tool / Component Palette do Delphi Win32), da sessão Data, arraste para o designer um componente GridView (o "DBGrid" do ASP.NET), para dentro da DIV. Repare que uma pequena seta surgirá no canto superior direito do controle, ela se chama Smart Tag e dá acesso às principais propriedades e itens passíveis de configuração para o controle (no Delphi Win32 esta opção aparece nas versões mais recentes do IDE, porém abaixo do Object Inspector).

Nota

Se não estiver visualizando a Tool Box, clique Ctrl + ALT + X.

Selecione Choose Data Source > New Data Source, a seguir escolha Database, no combo escolha nossa conexão configurada previamente no IDE, confirme a criação da Connection String (falaremos mais sobre isso adiante), marque o * para trazer todos os campos da tabela Products (**Figura 12**) e finalize o assistente. Ainda na Smart Tag, se desejar escolha uma auto-formatação para o controle. Aperte Ctrl+F5 (equivale ao Run Without Debugging do Delphi Win32) e veja o resultado no browser (**Figura 13**).

Conclusão

Com isso finalizamos a primeira etapa desta série. Como você pôde comprovar, criar bancos no SQL Server é tão simples quanto criar bancos no Firebird com IBExpert. Como você também já deve ter reparado com o primeiro contato com o Delphi Prism, o IDE do Visual Studio tem estrutura bastante similar ao RAD Studio. O desenvolvimento visual baseado em componentes é idêntico. Vamos comprovar isso ainda mais quando iniciarmos efetivamente o desenvolvimento do Web Site de E-Commerce. Lembre-se que agora seu sistema rodará nas nuvens, acessível a uma

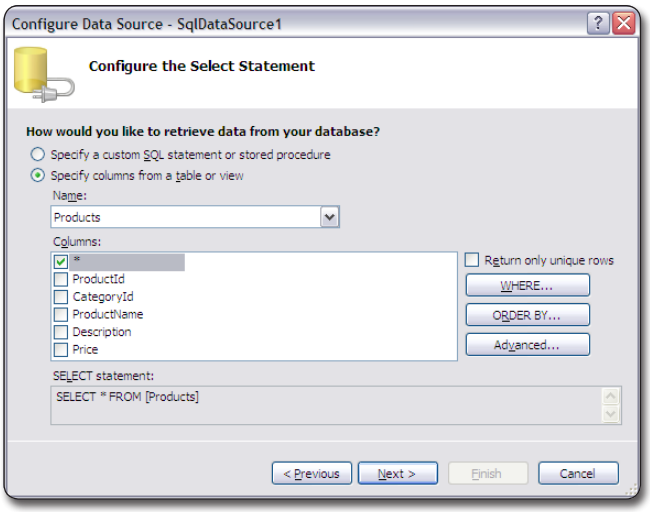


Figura 12. Assistente ajuda a configurar a conexão e construção da consulta SQL

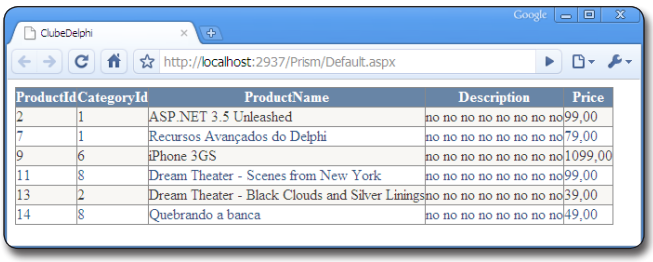


Figura 13. Teste Ok! Dados da tabela de produtos sendo visualizados no Google Chrome

gama muito maior de usuários, o leque de abrangência de seus clientes aumentará, problemas com manutenção, instalação, configuração, compatibilidade entre diferentes sistemas operacionais, tudo na Web é mais fácil. Bom para nós, bom para nossos clientes. Bem-vindo ao mundo do Delphi Prism. ●

Links

Delphi Prism
https://downloads.embarcadero.com/free/delphi_prism

Microsoft SQL Server Management Studio Express
http://www.microsoft.com/downloads/details.aspx?FamilyId=C243A5AE-4BD1-4E3D-94B8-5A0F62BF7796&displaylang=en

SQL Server Books OnLine
http://go.microsoft.com/fwlink/?LinkId=157385

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

XML

Persistência em XML com MVC

Do que trata o artigo
Este artigo trata da tecnologia SimpleXML, através de uma aplicação prática seremos capazes de exibir em tela dados contidos em um arquivo XML.

Para que serve
Este artigo serve para demonstrar como manipular documentos XML através da linguagem PHP, fazendo alteração dos dados contidos em um arquivo.

Em que situação o tema é útil
As técnicas apresentadas neste artigo serão úteis para sistemas que precisam promover a integração de informações entre diferentes plataformas, através da uma linguagem comum, a XML.

Lucas Simões Maistro
lucassimoes@univem.edu.br
Analista/Programador do Centro Universitário Eurípides de Marília. É desenvolvedor Delphi desde 1999 e trabalha com Oracle e SQL Server desde 2001. Graduado em Análise de Sistemas e Pós Graduado em Especialização em Sistemas para Internet pelo UNIVEM.

Raphael Zanon Rodrigues
zanonr@gmail.com
É Analista/Programador do Centro Universitário Eurípides de Marília. Desenvolvedor Delphi. Graduado em Análise de Sistemas e Pós Graduado em Especialização em Sistemas para Internet pelo UNIVEM.

Resumo do DevMan

Este artigo mostra como criar uma página PHP onde o usuário vai poder digitar dados sobre alunos, em uma tela de cadastro. Estes dados serão gravados em um arquivo XML, que poderão ser consultados em uma grid. Para isso, utilizaremos o arquivo XML como se fosse um banco de dados, através da metodologia MVC (Model – View – Controller).

Atualmente, o desenvolvimento de software exige que estejamos preparados a realizar integrações entre diversas plataformas, tecnologias e linguagens. Antes desenvolvíamos um software desktop restrito a apenas um sistema operacional, ou no desenvolvimento Web criávamos a aplicação para que funcionasse em alguns browsers. Hoje vivemos em um mundo onde um software não pode estar restrito a esse tipo de limitação, é necessário pensarmos em software que permita a adaptação para novas plataformas.

A partir desse cenário, entendemos também que, quando se trabalha com diversas plataformas, é perfeitamente possível que existam incompatibilidades entre elas, sendo assim tecnologias como XML e JSON (JavaScript Object

Notation) tomam força, uma vez que são implementadas por praticamente todas as plataformas atuais. Neste artigo vamos abordar o formato XML, deixando para uma outra oportunidade uma abordagem da tecnologia JSON com PHP.

XML é uma linguagem de marcação que consiste em uma série de regras que divide um documento em partes lógicas e hierárquicas, sendo na verdade um subconjunto da linguagem SGML (Standard Generalized Markup Language). SGML é um padrão internacionalmente aceito para descrever praticamente qualquer tipo de informação, no entanto, é demasiadamente complexo para a Web. Desta forma, o órgão World Wide Web Consortium (W3C) criou uma versão modificada do SGML especificamente para a Web, a XML.

A XML trabalha com tags descritivas de marcação, muito parecidas com o HTML. A diferença é que o HTML foi projetado especificamente para formatar dados para navegadores da Web e, como tal, é limitada a um conjunto pré-definido de tags e funções. Já o XML apenas estabelece as regras para a marcação, deixando que o autor do documento defina suas próprias tags.

A XML foi projetada para incorporar as seguintes funcionalidades:

• Facilidade de uso - Por força das marcas descritivas que contêm, os documentos XML são fáceis de ler e entender, mesmo para os usuários com pouco conhecimento na área de desenvolvimento de software ou mesmo computação. Eles também são extremamente simples de criar, um usuário não-técnico é normalmente capaz de criar um documento XML em menos tempo do que levaria para criar um documento HTML correspondente;

• Estrutura formal - Embora tenhamos uma enorme flexibilidade na atribuição de nomes das tags oferecidas pelo padrão XML e possamos utilizar nossas próprias tags, o padrão XML impõe uma estrutura formal de um documento. Palavras chave devem ser nomeadas e aninhadas corretamente, tags de abertura devem ter marcas de fechamento correspondente e namespaces devem ser definidos sempre que forem necessários. Estas regras asseguram que cada documento XML reúna algumas expectativas mínimas de estrutura e sintaxe, e torne mais fácil para as aplicações e os processadores lidar com dados XML;

• Suportado por muitos aplicativos - Como a XML é fácil de usar e rápido para se trabalhar, não é difícil escrever um aplicativo que use dados XML codificados. Uma série de analisadores XML está disponível on-line, editores de XML, validadores e ferramentas, além do fato de que os browsers mais populares suportam XML;

• Compatível com SGML - Conforme mencionado anteriormente, a XML é um desdobramento do SGML, e documentos XML cumprem todas as regras e restrições de marcação SGML. XML combina o poder da SGML com a flexibilidade e facilidade de uso que esperamos da Internet.

Existem algumas abordagens básicas para analisar um documento XML, uma delas é baseada em eventos, outra em árvore. O analisador de documentos Simple API for XML (SAX) utiliza a primeira abordagem para análise dos documentos XML, enquanto que o Document Object Model (DOM) utiliza a segunda abordagem. A diferença básica e mais notória é que o DOM carrega todo o documento para a memória para analisá-lo, enquanto o SAX faz uma leitura por demanda, em partes.

O DOM é uma interface padrão de acesso e manipulação de estruturas de dados e, como o nome sugere, essas funcionalidades acontecem através da modelagem de uma árvore hierárquica de objetos. Dessa forma, a principal característica do DOM é que ele carrega todo o documento para a memória e tem à disposição uma visão de todos os objetos na memória como numa árvore (tree-based). Para se navegar e manipular os dados dessa árvore existe uma grande quantidade de objetos no DOM que oferecem métodos e atributos que podem ser usados por uma aplicação para tal finalidade, como por exemplo, as classes DomDocument, DomElement etc.

Já o SAX é mais indicado quando o documento a ser analisado é muito extenso, pois carregá-lo todo na memória não é uma prática muito boa, seria mais interessante ir carregando partes do documento conforme a necessidade. Como nosso exemplo será criar uma aplicação que armazene e manipule os dados em um arquivo XML, utilizaremos essa API no projeto. O SAX tem um papel muito claro e definido, sua função é meramente analisar dados XML que estão sendo consumidos por ele e fornecer as funções mais apropriadas para a manipulação dos dados.

Para melhor compreensão de como o SAX funciona, veja a **Listagem 1**, ela mostra um arquivo XML com um registro de pedido de compra, ao analisar este documento o SAX deixaria rastros de eventos parecido com o que mostramos na **Listagem 2**.

Na **Listagem 1** temos um exemplo de arquivo XML que contém dados de pedidos. Como dito anteriormente, o arquivo XML é gerado de forma organizada e estruturada, permitindo assim que sejam compreendidos tanto por mecanismos de parses como por qualquer pessoa. De uma forma simples podemos notar que o arquivo XML possui apenas um pedido na lista (123415) que pertence a Jose Silva, morador de Marília em SP.

Nota

O termo "parse" se refere ao processamento / análise de um documento XML, feito geralmente por um "parser" (processador).

Já a **Listagem 2** mostra os eventos disparados pelo SAX à medida que o parse do arquivo estava sendo feito. Perceba que diferente do DOM, os dados e estrutura do arquivo não são mantidos na memória, no entanto precisamos saber qual a estrutura do XML assim como os dados. Isso é feito no SAX através de notificações. À medida que o SAX vai avançando no XML vai disparando eventos que nos informam o que está ocorrendo naquele momento.

No exemplo da **Listagem 2** podemos ver que ao iniciar o parser o SAX dispara um evento (linha 01). Isso nos informa

que a tag PEDIDOS foi aberta. O SAX continua avançando no XML e para cada ação um evento é disparado nos informando o que foi feito. Por exemplo, na linha 06 ocorre a leitura do valor "José". Observe que antes disso foram abertas as tags "nome" (linha 04), "primeiro" (linha 05) e depois elas foram fechadas, de acordo com a regra de abertura e fechamentos de tags da XML.

Listagem 1. Exemplo de um documento XML

```
<?xml version="1.0"?>
<PEDIDOS>
 <pedido> 123415
<nome>
    <primeiro>Jose</primeiro>
    <sobrenome>Silva</sobrenome>
</nome>
<endereco>
    <rua>Rua Alvares Cabral</rua>
    <numero>15</numero>
    <cidade>Marilia </cidade>
    <uf>SP</uf>
    <pais>BR</pais>
</endereco>
</pedido>
</PEDIDOS>
```

Listagem 2. Rastro de eventos feito pelo SAX ao analisar documento da **Listagem1**

```
01. [tag opened] PEDIDOS
02. [tag opened] pedido
03. [cdata] 123415
04. [tag opened] nome
05. [tag opened] primeiro
06. [cdata] Jose
07. [tag closed] primeiro
08. [tag opened] sobrenome
09. [cdata] Silva
10. [tag closed] sobrenome
11. [tag closed] nome
12. [tag opened] endereco
13. [tag opened] rua
14. [cdata] Rua Alvares Cabral
15. [tag closed] rua
16. [tag opened] numero
17. [cdata] 15
18. [tag closed] numero
19. [tag opened] cidade
20. [cdata] Marilia
21. [tag closed] cidade
22. [tag opened] uf
23. [cdata] SP
24. [tag closed] uf
25. [tag opened] pais
26. [cdata] BR
27. [tag closed] pais
28. [tag closed] endereco
29. [tag closed] pedido
30. [tag closed] PEDIDOS
```

A partir da versão 5.0 o PHP passa a contar com uma nova extensão para acesso a documentos XML, que tem por objetivo tornar mais simples a implementação de código de manipulação de documentos XML, o SimpleXML. Dentre suas características podemos destacar o fato dela converter um documento XML em um objeto, fazendo com que os elementos do XML sejam acessados em forma de propriedades usando notação de orientação a objetos simples. Uma grande característica é que quando o SimpleXML identifica uma série de elementos com o mesmo nome e nível, ele pode representar esses dados como um simples array, extremamente fácil de ser lido e manipulado. É o que comprovaremos neste artigo. Além disso, o SimpleXML pode trabalhar em conjunto com o DOM, através de importação.

Considerações iniciais do projeto

Para demonstrar como utilizar do SimpleXML no PHP iremos aproveitar o exemplo do artigo da edição 114, onde transformamos uma aplicação OO em uma aplicação que fazia uso do padrão MVC. Na ocasião criamos um cadastro de alunos que alimenta uma base de dados MySQL. Aqui iremos alterar os métodos de acesso a dados, de forma que esses acessem e manipulem um arquivo XML.

Crie uma pasta chamada XML_PHP no diretório do seu servidor Apache, crie dentro dela um arquivo chamado aluno.xml e codifique-o conforme a **Listagem 3**. Observe que este arquivo apenas define a estrutura para receber dados de um aluno, além de conter o cabeçalho XML padrão. Esse arquivo será manipulado posteriormente, onde serão inseridos dados de alunos, conforme o usuário digitará em tela.

Listagem 3. Arquivo aluno.xml

```
<?xml version="1.0" encoding="utf-8"?>
<aluno>
</aluno>
```

Nota

Para desenvolver e testar o exemplo do artigo, vamos utilizar o WampServer (veja sessão Links).

Camada View

Crie um novo arquivo com o nome aluno_view.class.php (**Listagem 4,** continuando na **Listagem 5**), esse arquivo conterá a classe Aluno_View, responsável pelos métodos de interface da aplicação, que em nosso exemplo serão os métodos formCadastro e formVisualizar. O código é composto por HTML mesclado com PHP.

Nota do DevMan

O padrão MVC consiste em dividir o projeto em três camadas lógicas: Model, View e Controller. A camada View é a camada responsável pelos mecanismos de interface, usada para receber a entrada de dados e apresentar o resultado. Já a camada Controller é composta pelas regras de negócio e faz uso dos mecanismos da classe Model e View, enquanto que a Model é a camada de acesso a dados responsável por fazer persistência e consultas de dados.

Na linha 4 iniciamos o método formCadastro, que tem como parâmetro esperado um array contendo um registro do arquivo XML. Veja nas linhas 18, 19, 26, 27, 34, 35, 42, 43, 50, 51, 58 e 59 que é feito um teste onde, caso haja valor no campo respectivo ao input criado, escrevemos a tag value e o valor retornado do array, criando assim um recurso de edição dos dados já cadastrados. Caso não existam valores, as caixas de texto aparecerão em branco prontas para serem preenchidas. As demais linhas são apenas código HTML para montar uma estrutura de tabela para dar forma aos controles que serão criados.

Continuamos o exemplo, logo abaixo do método formCadastro inserimos o método formVisualizar (**Listagem 5**), como pode ser visto a partir da linha 70. Inicialmente montamos um pequeno cabeçalho com códigos HTML onde serão exibidos os links Cadastrar e Visualizar (linhas 72 a 77). Na sequência criamos uma nova tabela HTML para exibir o cabeçalho dos registros (linhas 79 a 86). Observe que entre as linhas 88 e 100 é feito um loop com a lista de alunos e em seguida seu retorno é mesclado com as tag HTML de criação de linhas e colunas da tabela. Além das colunas representando cada campo do documento XML, foram colocadas mais duas colunas no final de cada linha que servirão para realizar as solicitações de edição e exclusão de registros (linhas 96 e 97).

Acessando o documento XML

O acesso ao documento XML será feito por meio da camada model conforme o padrão MVC exige, sendo assim crie um arquivo chamado aluno_model.class.php e digite o código da **Listagem 6** (que continua nas **Listagens 7** e **8**).

Observe que primeiramente declaramos os atributos que representam os dados armazenados no arquivo XML e em seguida criamos os métodos get e set para cada um deles (linhas 2 a 22). Entre as linhas 24 e 30 criamos o método construtor da classe, que atribui valor a cada um de seus atributos, obtidos através de $_REQUEST. Por exemplo, $_REQUEST["nome"] recupera o valor informado no formulário de cadastro para o campo "nome".

Listagem 4. Classe Aluno_View

```
1.  <?php
2.  class Aluno_View {
3.      function __construct( ) {              }
4.      public function formCadastro($linha) { ?>
5.          <table width="400" height="157" border ="3"
              align=center bgcolor="gray">
6.          <tr bgcolor="#AABBCC"><td align="center">
7.              <font color="white"><b><a href="index.php?metodo =
                  cadastro" style="text-decoration: none; color:#ffffff">
                  Cadastrar</a>
8.          </td><td align="center">
9.              <font color="white"><b><a href="index.php?metodo=
                  visualiza" style="text-decoration: none; color:#ffffff">
                  Visualizar</a>
10.         </td> </tr><tr>
11.         <td colspan="2" width="166">
12.         <table width="410" height="140" border="0" bgcolor=
              "#DDDDDD">
13.             <form method="POST" action="" name="form_cadastro">
14.         <tr>
15.         <td width="51">Identifica&ccedil;&atilde;o.: </td>
16.         <td width="120" align="left"><input type="text" name=
              "ID" size="10" readonly="true"
17.         <?php
18.             if (!empty($linha["id_aluno"]))
19.                 echo "value='".$linha["id_aluno"]."'";
20.         ?>
21.         ></td>
22.         </tr>
23.         <tr>  <td width="51">Nome.: </td>
24.         <td width="120"><input type="text" name="nome" size="50"
25.         <?php
26.             if (!empty($linha["NOME"]))
27.                 echo "value='".$linha["NOME"]."'";
28.         ?>
29.         ></td>
30.         </tr><tr>
31.         <td width="51">Endere&ccedil;o.: </td>
32.         <td width="120"><input type="text" name="endereco"
              size="50"
33.         <?php
34.             if (!empty($linha["ENDERECO"]))
35.                 echo "value='".$linha["ENDERECO"]."'";
36.         ?>
37.         ></td>
38.         </tr><tr>
39.         <td width="51">Bairro.: </td>
40.         <td width="120"><input type="text" name="bairro"
              size="50"
41.         <?php
42.             if (!empty($linha["BAIRRO"]))
43.                 echo "value='".$linha["BAIRRO"]."'";
44.         ?>
45.         ></td>
46.         </tr><tr>
47.         <td width="51">Cidade.: </td>
48.         <td width="120"><input type="text" name="cidade"
              size="50"
49.         <?php
50.             if (!empty($linha["CIDADE"]))
51.                 echo "value='".$linha["CIDADE"]."'";
52.         ?>
53.         ></td>
54.         </tr><tr>
55.         <td width="51">Email.: </td>
56.         <td width="120"><input type="text" name="email"
              size="50"
57.         <?php
58.             if (!empty($linha["EMAIL"]))
59.                 echo "value='".$linha["EMAIL"]."'";
60.         ?>
61.         ></td>
62.         </tr><tr>
63.         <td colspan="2" align="center"><input type="submit"
              value="Gravar"></td>
64.         </tr>
65.         </form>
66.         </table>
67.         </td></tr>  </table>
68.  <?php
69. }
```

Listagem 5. Método formVisualizar da classe Aluno_View

```
70. public function formVisualizar($lista_alunos){       ?>
71. <table width="1000" border ="2" align=center bgcolor="gray">
72.       <tr bgcolor="#AABBCC">
73.       <td align="center">
74.       <font color="white"><b><a href="index.php?
            metodo=cadastro" style="text-decoration: none;
            color:#ffffff">Cadastrar</a>
75.       </td><td align="center">
76.       <font color="white"><b><a href="index.php?
            metodo=visualiza" style="text-decoration: none;
            color:#ffffff">Visualizar</a>
77.       </td> </tr><tr>
78.       <table width="1000" border ="2" align=center bgcolor="gray">
79.       <tr bgcolor="#AABBCC">
80.       <td width="25"  align="center" style="color:#ffffff">ID</td>
81.       <td width="100" align="center" style="color:#ffffff">Nome</td>
82.       <td width="100" align="center" style="color:#ffffff">
            Endere&ccedil;o</td>
83.       <td width="75"  align="center" style="color:#ffffff">Bairro</td>
84.       <td width="50"  align="center" style="color:#ffffff">Cidade</td>
85.       <td width="25"  align="center" style="color:#ffffff">Email</td>
86.       <td></td>  <td></td> </tr>
87. <?php
88.       foreach ($lista_alunos->dados as $linha) { ?>
89.             <tr bgcolor="#DDDDDD">
90.               <td align="center"> <?php echo $linha["id_aluno"] ?></td>
91.               <td> <?php echo $linha["NOME"]    ?></td>
92.               <td> <?php echo $linha["ENDERECO"] ?></td>
93.               <td> <?php echo $linha["BAIRRO"]   ?></td>
94.               <td> <?php echo $linha["CIDADE"]   ?></td>
95.               <td> <?php echo $linha["EMAIL"]    ?></td>
96.               <td width="25"  align="center" style="color:#ffffff">
                  <?php echo "<a href='index.php?metodo=
                  cadastro&ID=".$linha["id_aluno"]."" ?>" style=
                  "text-decoration: none; color:blue">Editar</a></td>
97.               <td width="25"  align="center" style="color:#ffffff">
                   <?php echo "<a href='index.php?metodo=visualiza&ID=
                    ".$linha["id_aluno"]."" ?> style="text-decoration: none;
                    color:black">Excluir</a></td>
98.             </tr> <?php
99.           }
100.       ?>
101.    </table></tr></table> <?php
102.         }
103.   }
104. ?>
```

Listagem 6. Classe Aluno_Model

```
1.  <?php
2.  class Aluno_Model {
3.        var $id_aluno;
4.        var $nome;
5.        var $email;
6.        var $endereco;
7.        var $bairro;
8.        var $cidade;
9.
10.       public function get_id_aluno()  { return $this->id_aluno;  }
11.       public function get_nome()      { return $this->nome;      }
12.       public function get_email()     { return $this->email;     }
13.       public function get_endereco()  { return $this->endereco; }
14.       public function get_bairro()    { return $this->bairro;    }
15.       public function get_cidade()    { return $this->cidade;    }
16.
17.       public function set_id_aluno($id_aluno)
          {$this->id_aluno = $id_aluno;       }
18.       public function set_nome($nome)
          { $this->nome = $nome;              }
19.       public function set_endereco($endereco)
          { $this->endereco = $endereco;    }
20.       public function set_bairro($bairro)
          { $this->bairro = $bairro; }
21.       public function set_cidade($cidade)
          { $this->cidade = $cidade;   }
22.       public function set_email($email)
          { $this->email = $email;  }
23.
24.       function __Construct() {
25.       $this->set_nome($_REQUEST["nome"]);
26.       $this->set_endereco($_REQUEST["endereco"]);
27.       $this->set_bairro($_REQUEST["bairro"]);
28.       $this->set_cidade($_REQUEST["cidade"]);
29.       $this->set_email($_REQUEST["email"]);
30.      }
31.
```

Listagem 7. Função para inserir aluno

```
32.  public function insert_Aluno($nome,$endereco,$bairro,$cidade,
     $email){
33.       $xml = simplexml_load_file("aluno.xml");
34.       $result = $xml->xpath("dados[last()]");
35.       foreach ($result as $sendId) {   $recId = $sendId['id_aluno']; }
36.            $id_aluno= $recId+1;
37.            $dados = $xml->addChild('dados');
38.            $dados->addAttribute('id_aluno', $id_aluno);
39.            $dados->addAttribute('NOME', $nome);
40.            $dados->addAttribute('ENDERECO', $endereco);
41.            $dados->addAttribute('BAIRRO', $bairro);
42.            $dados->addAttribute('CIDADE', $cidade);
43.            $dados->addAttribute('EMAIL', $email);
44.            $status = file_put_contents('aluno.xml',$xml->asXML() );
45.            return $status;
46.  }
```

Nota

Lembre-se que o PHP é case sensitive, ou seja, ele diferencia letras maiúsculas de minúsculas. Assim o código $_REQUEST["nome"] é completamente diferente de $_REQUEST["Nome"].

Na linha 32, já na **Listagem 7**, declaramos o método insert_Aluno, ele tem a responsabilidade de inserir registros no arquivo XML. Veja que na linha 33 instanciamos um objeto através do uso da função simplexml_load_file, informando como parâmetro o arquivo XML criado anteriormente. Essa função é utilizada para converter um arquivo XML em um objeto. Na linha seguinte utilizamos o método XPath do objeto $xml para realizar uma consulta nos dados do arquivo que acabou de ser carregado. O retorno desse método é um array contendo os dados localizados na pesquisa.

O XPath é uma linguagem utilizada para localizar parte de um documento XML. Por exemplo, ao parametrizarmos a função XPath com a expressão dados[last()] (linha 34), solicitamos que a função busque no objeto $xml o último registro da tag dados, que ainda não existe no arquivo aluno.xml original mas que será criada sempre que um aluno novo for cadastrado. O motivo disso é incrementar o valor do id_aluno para que o usuário não tenha a necessidade de informá-lo ao realizar o cadastro.

Para realizar a função de incremento, deve-se utilizar a função foreach, pois mesmo contendo um único registro, precisamos armazenar o valor encontrado em uma variável, que na linha 36 é incrementada. Na sequência fazemos uso de outro método do objeto $xml, o addChild (linha 37). Esse método é responsável por criar um novo registro da tag dados dentro do objeto $xml, que ao final será devolvido para o documento XML correspondente. Veja que entre as linhas 38 e 43 são definidos os atributos da tag dados, que correspondem a cada atributo da classe aluno.

E por fim, devolvemos o resultado de todas essas alterações ao arquivo aluno.xml com a função do PHP file_put_contents (linha 44). Essa função funciona como se executássemos as funções fopen, fwrite e fclose sucessivamente. Como parâmetros informamos o nome do arquivo a ser gravado e o conteúdo que queremos gravar, que neste caso virá do objeto $xml através da função asXML.

Após programarmos o método insert_Aluno, vamos implementar o método responsável por executar alterações nos dados do documento XML, para isso usaremos o método descrito na **Listagem 8**. Primeiramente observe que neste método devemos passar como parâmetro todos os atributos que compõem a classe aluno, pois a localização do registro a ser alterado ocorrerá a partir da identificação do aluno.

A linha 49 tem a mesma função da linha 33 do método insert_Aluno. Também usamos XPath, porém dessa vez veja que a expressão informada é dados[@id_aluno=$id_aluno], que busca no objeto $xml um registro onde o atributo id_aluno seja igual ao valor do primeiro parâmetro informado.

Entre as linhas 51 e 58 utilizamos novamente a função foreach para manipular o array de registros devolvido pelo método XPath, porém agora atribuímos a cada atributo desse array os valores passados por parâmetro ao método. Por fim, nas linhas 58 e 59 efetuamos o mesmo procedimento realizado no método da **Listagem 7**.

O método delete_Aluno (**Listagem 9**) é o mais simples dos descritos até aqui, pois após instanciar o documento XML completo, verifica cada um dos registros até que encontre o que tem a identificação do aluno informado por parâmetro. Uma vez encontrado o registro, executamos a função unset (linha 67) nativa do PHP, que vai destruir da memória a variável informada, que nesse caso é representada pelo item zero do array dados. Ao final desse método devolvemos os dados do objeto $xml para o documento XML (linhas 71 e 72).

Listagem 8. Função para atualizar aluno

```
47.  public function update_Aluno($id_aluno,$nome,$endereco,
     $bairro,$cidade,$email){
48.      $xml = simplexml_load_file("aluno.xml");
49.      $result = $xml->xpath("dados[@id_aluno=$id_aluno]");
50.      foreach ($result as $indice => $dados)
51.          {
52.           $dados['NOME']  = $nome;
53.           $dados['ENDERECO']              = $endereco;

54.           $dados['BAIRRO'] = $bairro;
55.           $dados['CIDADE'] = $cidade;
56.           $dados['EMAIL'] = $email;
57.          }
58.      $status = file_put_contents('aluno.xml',$xml->asXML() );
59.     return $status;
60.  }
```

Listagem 9. Função para excluir aluno

```
61.  public function delete_Aluno($id_aluno){
62.      $xml = simplexml_load_file("aluno.xml");
63.      foreach ($xml as $dados)
64.      {
65.          if ($dados['id_aluno'] == $id_aluno)
66.          {
67.              unset ($dados[0]);
68.              break;
69.          }
70.      }
71.      $status = file_put_contents('aluno.xml',$xml->asXML() );
72.      return $status;
73.  }
```

Para finalizar a construção da classe aluno_model, devemos preparar um método de consulta (**Listagem 10**). Na linha 75 recuperamos o valor do atributo id_aluno da classe Aluno para dentro de uma variável, em seguida instanciamos o documento XML (linha 76) e verificamos se a variável id_aluno tem algum valor (linha 77), caso não tenha, retornamos como resultado do método todo o conteúdo do documento XML (linha 83). No caso de alguma identificação de aluno tenha sido encontrada, realizamos uma pesquisa no objeto $xml através de seu método XPath (linha 79) e informamos o resultado dessa pesquisa ao resultado do método (linha 80).

Camada Controller

Vamos agora implementar a classe Aluno_Controller que fará o "meio campo" entre as camadas model e view e

Listagem 10. Função para consultar alunos

```
74.     public function consultar_Alunos(){
75.         $id_aluno = $this->get_id_aluno();
76.         $xml = simplexml_load_file("aluno.xml");
77.         if ($id_aluno != '')
78.         {
79.           $result = $xml->xpath("dados[@id_aluno=$id_aluno]");
80.           return $result;
81.         }
82.         else
83.           return $xml;
84.     }
85. }
86. ?>
```

disponibilizará seus resultados à página principal de nossa aplicação. Na camada controller ocorrem todas as solicitações aos métodos das classes Aluno_Model e Aluno_View.

Crie um arquivo chamado aluno_controller.class.php e digite o conteúdo das **Listagens 11** e **12**. A classe Aluno_Controller contém apenas dois atributos e quatro métodos. Os atributos são na verdade uma instância de Aluno_Model e uma instância de Aluno_View, conforme pode ser visto das linhas 6 a 12. As instâncias são criadas no método construtor da classe. Já os métodos declarados, cad_Aluno, atualiza_Aluno, visualiza_Aluno e exclui_Aluno realizam as operações básicas de manutenção de dados.

Listagem 11. Classe Aluno_Controller

```
1.  <?php
2.  require_once("aluno_model.class.php");
3.  require_once("aluno_view.class.php");
4.
5.  class Aluno_Controller {
6.      var $mdl;
7.      var $vw;
8.
9.      public function __Construct(){
10.         $this->mdl = new Aluno_Model();
11.         $this->vw = new Aluno_View();
12.     }
13.
14.     public function cad_Aluno() {
15.         $nome = $this->mdl->get_nome();
16.         $endereco = $this->mdl->get_endereco();
17.         $bairro = $this->mdl->get_bairro();
18.         $cidade = $this->mdl->get_cidade();
19.         $email = $this->mdl->get_email();
20.
21.         if (!empty($nome) && !empty($endereco) && !empty
            ($bairro) && !empty($cidade) && !empty($email)) {
22.             $status = $this->mdl->insert_Aluno($nome,
                $endereco,$bairro,$cidade,$email);
23.             if ($status) {
24.                 $lista_alunos = $this->mdl->consultar_Alunos();

25.                 $this->vw->formVisualizar($lista_alunos);
26.                 echo "<br><center><b>Aluno Cadastrado com
                    sucesso.</b></center><br>";
27.             }
28.             else
29.                 echo "<br><center><b>N&atilde;o foi poss&iacute;vel
                    inserir os dados.</b></center><br>";
30.     }
31.     else {
32.         $id_aluno = $this->mdl->get_id_aluno();
33.         if (!empty($id_aluno)) {
34.             $aluno = $this->mdl->consultar_Alunos();
35.             foreach ($aluno as $dados => $linha);
36.         }
37.         $this->vw->formCadastro($linha);
38.     }
39. }
```

Listagem 12. Atualiza aluno, da classe Aluno_Controller

```
40.  public function atualiza_Aluno($id) {
41.      $this->mdl->set_id_aluno($id);
42.
43.      $nome = $this->mdl->get_nome();
44.      $endereco = $this->mdl->get_endereco();
45.      $bairro = $this->mdl->get_bairro();
46.      $cidade = $this->mdl->get_cidade();
47.      $email = $this->mdl->get_email();
48.
49.      if (!empty($nome) && !empty($endereco) && !empty
         ($bairro) && !empty($cidade) && !empty($email)) {
50.          $status = $this->mdl->update_Aluno($id,$nome,
             $endereco,$bairro,$cidade,$email);
51.          if ($status) {
52.              $this->mdl->set_id_aluno("");
53.              $lista_alunos = $this->mdl->consultar_Alunos();
54.              $this->vw->formVisualizar($lista_alunos);
55.              echo "<br><center><b>Dados Atualizados com sucesso.
                 </b></center><br>";
56.          }
57.          else
58.              echo "<br><center><b>N&atilde;o foir poss&iacute;vel
                 atualizar os dados.</b></center><br>";
59.          }
60.          else {
61.              $id_aluno = $this->mdl->get_id_aluno();
62.              if (!empty($id_aluno)) {
63.              $aluno = $this->mdl->consultar_Alunos();
64.                  foreach ($aluno as $dados => $linha);
65.          }
66.          $this->vw->formCadastro($linha);
67.      }
68.  }
69.
70.      public function visualiza_Aluno() {
71.      $lista_alunos = $this->mdl->consultar_Alunos();
72.      $this->vw->formVisualizar($lista_alunos);
73.     }
74.
75.      public function exclui_Aluno($id) {
76.      $this->mdl->set_id_aluno($id);
77.
78.      $status = $this->mdl->delete_Aluno($id);
79.      if ($status) {
80.          $this->mdl->set_id_aluno("");
81.          $lista_alunos = $this->mdl->consultar_Alunos();
82.          $this->vw->formVisualizar($lista_alunos);
83.          echo "<br><center><b>Registro exclu&iacute;do com
             sucesso.</b></center><br>";
84.      }
85.      else
86.          echo "<br><center><b>N&atilde;o foir poss&iacute;vel
             realizar a exclus&atilde;o.</b></center><br>";
87.      }
88.      }
89. ?>
```

O método cad_Aluno inicia recuperando os dados do objeto aluno que está instanciado (linhas 15 a 19). Feito isso fazemos a verificação para garantir que todos os dados tenham sido passados e caso a afirmação seja verdadeira executamos o método insert_Aluno (linha 22). Caso contrário este verifica se esse registro já existe no documento XML e carrega a tela de cadastro com os dados encontrados (linhas 21 a 38). Ainda sobre o método cad_Aluno, caso o método insert_Aluno seja executado com sucesso, então executamos o método consultar_Alunos que retorna o array com todos os alunos cadastrados (linha 24). Por fim passamos este array para o método formVisualizar que se encarrega de montar a tabela com os dados (linha 25).

A partir da linha 40 (**Listagem 12**) está o código do método atualiza_Aluno, responsável por atualizar os dados após a edição. Esse método é muito parecido com o método cad_Aluno, com a diferença de que o registro necessariamente já existe. Entre as linhas 70 e 88 estão os métodos visualiza_Aluno e exclui_Aluno, que por sua simplicidade dispensam maiores explicações.

Página Inicial

Por fim criaremos o arquivo index.php que é a nossa página inicial, como pode ser visto na **Listagem 13**. O único detalhe a ser observado é a tag HTML "meta" (linha 05), que define o padrão de caracteres utf-8, padrão esse que faz com que consigamos trabalhar com acentuação e caracteres especiais.

A única coisa que fazemos aqui é instanciar a classe aluno_Controller (linha 13) e na sequência verificamos se foi passado à página index.php algum parâmetro (linha 14). Se vazio (ou for "cadastro"), vamos inserir um novo aluno (linha 16), se passado um ID, faremos uma atualização (linha 19). Caso o parâmetro passado à página não seja "cadastro" então estaremos ou querendo visualizar todos os registros (linha 24) ou deletar um existente (linha 27).

Testando

Chegou a hora de testar a aplicação. Execute o script index.php via browser utilizando um diretório virtual do Apache, que pode ser o raiz (veja nota). A **Figura 1** mostra a aplicação em execução.

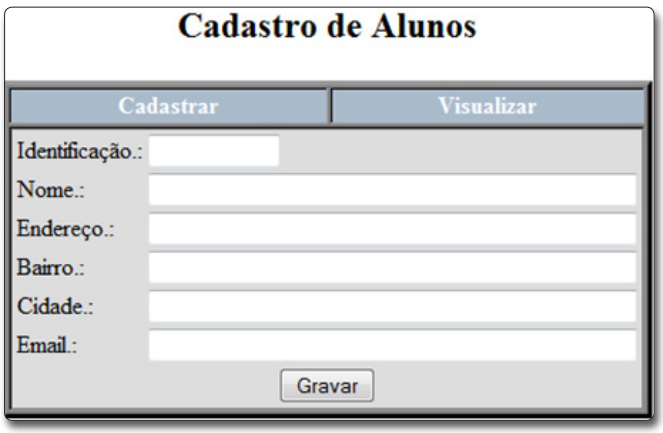


Figura 1. Aplicação em execução

Nota

Para testar a aplicação basta copiar os scripts php criados neste exemplo mais o arquivo aluno.xml para a pasta root do WampServer, que na instalação padrão fica em c:\wamp\www. Feito isso basta acessar via browser o endereço HTTP://localhost/index.php

Realize algumas operações de inclusão, alteração e exclusão e observe que os dados são inseridos e vão sendo

Listagem 13. Página inicial

```
01. <?phprequire_once("./aluno_controller.class.php"); ?>
02. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional
    //EN""http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
03. <html>
04. <head>
05. <meta http-equiv="Content-Type" content="text/html;
    charset=utf-8" />
06. </head>
07.     <body onLoad="">
08.     <center>
09.     <h2>Cadastro de Alunos</h2>
10.     <table border=0>
11.         <tr><td>
12.             <?php
13.             $aluno = new Aluno_Controller( );
14.             if (($_REQUEST["metodo"] == "cadastro") || (empty
                ($_REQUEST["metodo"]))) {
15.                 if (empty($_REQUEST["ID"])) {
16.                     $aluno->cad_Aluno();
17.                 }
18.                 else {
19.                     $aluno->atualiza_Aluno($_REQUEST["ID"]);
20.                 }
21.             }
22.             else {
23.                 if (empty($_REQUEST["ID"])) {
24.                     $aluno->visualiza_Aluno();

25.                 }
26.                 else {
27.                     $aluno->exclui_Aluno($_REQUEST["ID"]);
28.                 }
29.             }
30.             ?>
31.         </td></tr></table>
32.     </center>
33.     </body>
34. </html>
```

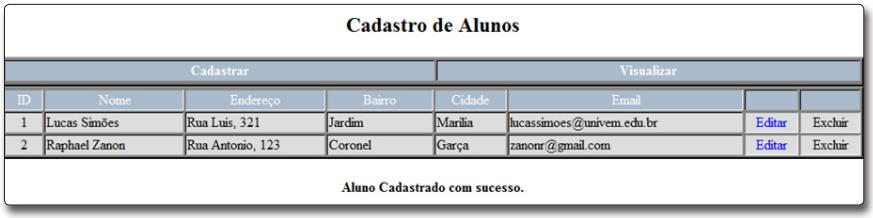


Figura 2. Listando os registros cadastrados

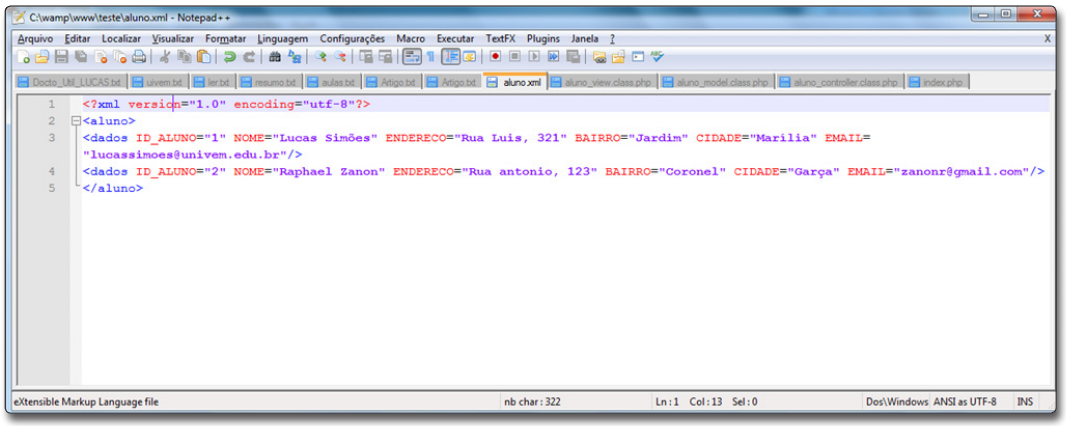


Figura 3. Documento XML

armazenados no documento XML aluno.xml. A **Figura 2** mostra os dados cadastrados sendo listados no browser. Abra o arquivo em um editor de texto ou caso você possua algum software de manipulação de documentos XML utilize-o e veja que ele ficará parecido com a **Figura 3**.

Conclusão

Pudemos ver neste artigo como é fácil desenvolver uma aplicação com o conceito de MVC para manipulação de arquivos XML no PHP. Vimos os conceitos de XML e as intenções com que suas funcionalidades foram projetadas. Vimos também o conceito de três interfaces de manipulação de estruturas de dados, sendo elas, DOM, SAX e SimpleXML, utilizando a última para o nosso exemplo. ●

Links

Site Oficial do WampServer
http://www.wampserver.com

XML Path Language
http://www.w3.org/TR/xpath/

Twitter dos Autores
http://twitter.com/lucas_simoes - http://twitter.com/zanonr

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Chegou o Sistema de Créditos DevMedia

Agora o **conteúdo completo** do nosso site está **ao seu alcance!**

A partir de agora todas as **4000 vídeo-aulas** do site DevMedia podem ser compradas individualmente.

Economia - Você não precisa mais assinar oito produtos diferentes para ter acesso ao conteúdo completo do site!

Todas as vídeos que você sempre quis ver a partir **R$ 0,99!**

Saiba mais sobre o Sistema de Créditos!

DevMedia GROUP

www.devmedia.com.br/creditos

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.